Coding Without Your Crystal Ball: Unanticipated Object-Oriented Reuse

Donna Malayeri

CMU-CS-09-163

December 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich, Chair
William Scherlis
Karl Crary
Todd Millstein, UCLA

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported in part by Aldrich's NSF CAREER award CCF-0546550, DARPA grant HR0011-0710019, and Army Research Office grant DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems."

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **DEC 2009**	2. REPORT TYPE	3. DATES COVERED **00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE **CodingWithout Your Crystal Ball: Unanticipated Object-Oriented Reuse**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University ,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**176**	

Keywords: Structural subtyping, nominal subtyping, external dispatch, multiple dispatch, multiple inheritance, inheritance diamond

Abstract

In many ways, existing languages place unrealistic expectations on library and framework designers, allowing some varieties of client reuse only if it is explicitly—sometimes manually—supported. Instead, we should aim for the ideal: a language design that reduces the amount of prognostication that is required on the part of the original designers. In particular, I show that languages can and should support a *combination* of structural and nominal subtyping, *external dispatch*, and a form of *multiple inheritance*.

Structural subtyping, which allows new types to be added to an existing hierarchy post-hoc, has been studied for decades, but a naïve combination of structural subtyping and external dispatch poses serious typechecking issues. Instead, I present a novel combination of structural subtyping, nominal subtyping, and external dispatch—external dispatch allowing programmers to write new code that dynamically dispatches on an existing hierarchy. In its absence, programmers will often resort to writing manual dispatch code, which is tedious, error-prone, and lacks extensibility.

External dispatch is also difficult to combine with another useful language feature—multiple inheritance. It so happens that *any* form of multiple inheritance (even Java-style) makes modular typechecking of external methods extremely difficult; this is due to the so-called "diamond problem." To sidestep these issues, I propose a novel form of multiple inheritance which does not allow diamonds, but recovers expressiveness through a generalized form of self-types.

Finally, since languages with structural subtyping are used mainly in the research community, it had thus far remained unclear whether structural subtyping is actually useful in practice. To answer this question, I performed a novel empirical study of existing Java programs, which found that (a) even nominally-typed programs could benefit from structural subtyping, and (b) there is a potential synergy between structural subtyping and external dispatch.

Acknowledgments

I would like to thank my parents for their love and support, and for always believing I could finish, even when I doubted it. My sister Leila has also been incredibly supportive and I owe a lot to her.

My advisor, Jonathan, has been a great help and inspiration to me over the years. His advising was continually appropriate to the situation—helping along when I needed it, and pushing me to do things on my own when I was ready. I believe I can honestly say that I could not have finished this dissertation without his support and guidance. I would also like to thank my former advisor, Bob Harper, both for giving me a good foundation in type theory and for believing I could do great things.

My thesis committee, Bill Scherlis, Karl Crary, and Todd Millstein, have also been a great help with technical matters; their feedback has greatly improved the quality of this work. Jeannette Wing has always been of great moral support; her enthusiasm and unwavering confidence in my abilities helped tremendously.

My friends and fellow grad students have helped to keep me (mostly) sane throughout the years and many of them have helped in technical matters as well. William Lovas[‡] has been a great help both as a friend and as a colleague, and has helped me get un-stuck many a time! I'd like to thank all the members of the Plaid group, particularly Neel Krishnaswami (who is double-plus good at type theory!), Kevin Bierhoff, Ciera Jaspan, Nels Beckman, Thomas LaToza, and Josh Sunshine. I'd also like to thank the students in the PoP group, who politely put aside their distaste for objects and had reasonable and rational discussions with me.

I'd like to thank my friends for their support, including Valerie Pomerantz, James Hendricks, Mark Palatucci (who encouraged me not to leave grad school, during the Pittsburgh Lindy Exchange!), Steve Matuszek, Danielle St. Cyr, Adam Goode, Corina Bardasuc, and the Pittsburgh Lindy Hop community. Cyrus McCandless helped come up with the clever title, Geoff Washburn helped with fonts and formatting (especially the I and the parens `( )` in the code font), and George Fairbanks and Shan Shan Huang gave great "citation" advice. Finally, Jitters coffee shop gave me a place to work in a relaxing and welcoming atmosphere.

Please enjoy this document, and see if you can find the Easter eggs! (Hint: you will be financially rewarded.)

[‡] Who likes double daggers.[‡]

Contents

List of Figures

List of Tables

List of Theorems and Lemmas

Chapter 1

Introduction

> "Where shall I begin, please your Majesty?" he asked.
> "Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."
>
> Lewis Carroll (*Alice's Adventures in Wonderland*)

1.1 Overview

Existing programming systems are often adequate for developing software, but can lack support for effectively evolving that software. For example, modifications to a program may require changing the source code of some library or framework, a library or framework whose maintainer is some other organization or team. Even if the library/framework is open-source, it is evolving as well—for bug fixes, if nothing else. As a consequence, local modifications to this source code must be manually updated when a new version of the library or framework is released. Unless both the local modifications and the changes in the new release are minor and localized, this approach is impractical.

One way to improve this situation is to create additional programming language support for software extensibility and code reuse. Using the wrong language can create obstacles for code reuse; some varieties of changes are only possible if they are explicitly supported by the code's design. This in turn requires that the need for the change be anticipated, which is unrealistic. Instead, a language design should aim to reduce the amount of prognostication that is required on the part of the original designers.

Object-oriented languages provide many mechanisms for unanticipated code reuse; for instance, programmers may code against an abstract interface (allowing new implementations to be substituted) or may use inheritance to reuse existing code. However, existing languages place restrictions on these constructs (or forgo static type safety), reducing their potential utility.

In particular, I argue that languages should support *retroactive abstraction*, *external dispatch*, and a form of *multiple inheritance*. The potential utility of each of these features is described at a high level in this chapter and with more detail in each corresponding chapter. In particular, Sect. 3.6 and Chapter 4 present real-world examples and empirical data, respectively, as concrete evidence to support the claim.

Assumptions and Definitions

Throughout this dissertation, I make the key assumption that *modularity* (in particular, modular typechecking) is essential to any sensible language design. A typechecking algorithm is modular if no link-time typechecking is required.[1] This property is critical in an runtime environment that permits dynamic code loading (e.g., DLLs, dynamic classloading), as such a static linking phase does *not* exist in these situations.

Additionally, I assume that *more* static checking is preferable to less; i.e., catching additional errors at compile time is a good thing. In particular, I would not consider "dynamically-typed" languages as providing a solution to the problems addressed in this dissertation.

In this document, I use the term *interface* to denote a set of methods along with their types. Where such a distinction is relevant, I use the term *nominal interface* in reference to languages with nominal subtyping. Note that this definition of "interface" differs from that of Java or C#, where (nominal) interfaces also have an associated tag that can be used in dispatch (via instanceof tests or reflection). We refer to such entities as "tagged interfaces" in cases where the distinction is pertinent. Note that "tagged" interfaces are necessarily also nominal interfaces.

1.2 Limitations of Existing Languages

This section describes, through example, limitations of existing languages that can negatively impact code reuse. Though the problems are not all directly related to one another, there are interactions between them.

Retroactive Abstraction

Sometimes, programmers wish to code against a particular *implicit* interface that is shared by two classes, so that objects of either class can be used. Unfortunately, the lack of retroactive abstraction in traditional languages makes it difficult accomplish this task.

Concretely, suppose we have a Java graphics drawing library with interface Drawable and classes Circle and Icon (Figure 1.1). Circle implements Drawable, but Icon implements no interfaces. In particular, it does not implement Drawable, since it does not support setting an alpha transparency.

Now, suppose we wish to write a method centerAndDraw that takes an object that supports the setPosition and draw methods, and draws an item centered on the canvas. In principle, this method should be applicable to either an Icon or a Circle. Unfortunately, no appropriate type exists that would allow instances of either Icon or Circle to be passed to the method. The programmer's only options are to use reflection (which is not statically type-safe), to create two identical versions of centerAndDraw (one that applies to Icon and one that applies to Drawable), or to use Object as the type of item and perform instanceof tests (also not statically type-safe).

[1] Obviously, there is a continuum of "modularity," with my definition of "modular" at one end, and whole-program analysis on the other. Millstein [Millstein and Chambers 2002; Millstein 2003] identifies several interesting points in this design space, in the context of external methods and multimethods.

```
interface Drawable {
   void draw();
   void setPosition(int x, int y);
   void setAlpha(int alpha);
}

class Circle implements Drawable {
   void draw() { ... }
   void setPosition(int x, int y) { ... }
   void setAlpha(int alpha) { ... }
}

class Icon {
   void draw() { ... }
   void setPosition(int x, int y) { ... }
}

void centerAndDraw(_____ item) { // what type to use here?
   ...
   item.setPosition(xpos, ypos);
   item.draw();
}
```

Figure 1.1: An appropriate type does not exist for the parameter to centerAndDraw.

Of course, if we did control all the code for the graphics library, there would be a simple solution: we would simply create a type Bitmap containing the two methods in question, and make Drawable extend Bitmap and Circle implement Bitmap. The type of centerAndDraw's argument would then be Bitmap. Unfortunately, in our scenario, only the maintainer of the graphics library would be able to make such a change.

The problem here is that Java does not support retroactive interface implementation, where a class could be declared as implementing an interface (or an interface be declared as extending another interface) after the point at which the class or interface was originally defined. This problem is not confined to Java, however—it arises in *any* nominally-typed language that supports modular typechecking.[2] Since modular typechecking is crucial for effective software development (particularly in a team environment), a practical solution must not preclude its possibility.

In my empirical studies (Chapter 4), I found that the aforementioned situation does indeed arise in practice: sometimes an appropriate type does not exist and programmers resort to code duplication and instanceof tests. Not only does this cause problems when a change must be

[2] Some have proposed nominally-typed languages that support a form of retroactive abstraction, but these designs are either awkward, have unusual semantics, or require non-modular typechecking [Wehr et al. 2007; Ostermann 2008]. These designs are described further in Sections 2.6 and 5.1.

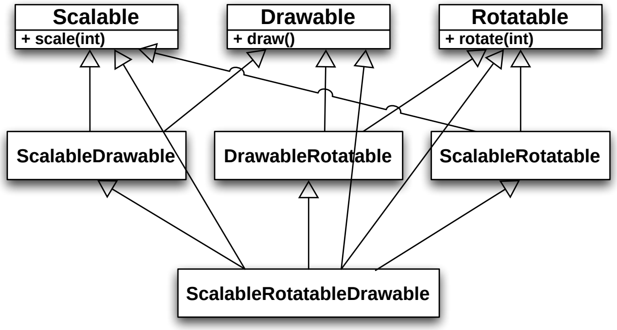

Figure 1.2: Manually composing interfaces using nominal subtyping

made to all of the code copies, the solution is not extensible; if a new type is later added that supports the required interface, programmers must add new code to handle the new type.

Composing Interfaces

There is another limitation of nominal subtyping: it makes it difficult to compose interfaces. Concretely, suppose that in our graphics library we have three interfaces: Scalable, Drawable, and Rotatable (Fig. 1.2). Now, if we wish to describe types that support some combination of these interfaces, nominal subtyping would require us to create 4 new interfaces. Aside from the tedious nature of this design, the programmer would also have to remember to use these compound interfaces appropriately. That is, if we have the declaration

```
class Glyph implements Scalable, Drawable
```

then a Glyph could not be passed to a method that expected a ScalableDrawable object, since Scalable, Drawable is not equivalent to ScalableDrawable. The root of the problem is that there are no type equalities in a system with only nominal subtyping; each new type name is distinct from all other types.

It would be possible to use *intersection types* to solve this particular problem [Coppo and Dezani-Ciancaglini 1978; Coppo et al. 1979; Pottinger 1980; Büchi and Weck 1998]; programmers would never create the composition interfaces and would instead write code in terms of e.g. Scalable $\wedge$ Drawable. However, this is not a complete solution; without explicit support for retroactive abstraction, programmers must still anticipate every possible “interesting” interface. In other words, intersection types would not solve the problem we saw above with the centerAndDraw method.

To solve both the problems of retroactive abstraction and composing interfaces, I propose using structural subtyping, which I describe in detail below.

Adding Methods to Existing Classes

Just as it is useful to add new types to an existing hierarchy, it can also be useful to add new code that operates on an existing hierarchy. Unfortunately, traditional languages make it difficult

```
static void readLine(Reader r) {
   if (r instanceof BufferedReader)
      ...    // look in buffer first
   else if (r instanceof StringReader)
      ...    // search string directly
   else if (r instance of InputStreamReader)
      ...    // read an array of bytes
   else
      ...    // call read() in a loop
}
```

Figure 1.3: Manual dispatch using instanceof tests

either to retroactively add new methods to existing classes, or to write new code that dispatches on an existing class hierarchy.

Concretely, suppose we wish to add new functionality for objects of type java.io.Reader. This class has a method for reading a byte at at time, but we would like to create a method that reads an entire line at a time.

Depending on the type of Reader object that we have, this readLine method would be implemented differently. A BufferedReader, for instance, can implement this method efficiently by searching for a newline character in the contents of the buffer, while a StringReader can perform an even more efficient operation. For other types of readers, we may perhaps implement this method by calling read(), which reads one byte at a time.

However, since Reader is part of the Java Standard Library, new methods cannot be added to it. Consequently, the only way to write new code that dispatches on this hierarchy is to hand-code dispatch using instanceof tests, as in Fig. 1.3. This design has several problems: it is tedious and error-prone (for example, cases for subtypes must appear before cases for supertypes) and it is not extensible. If another developer later adds a new subclass of Reader, a new case must be added to readLine—posing problems if this developer cannot modify readLine.

Of course, if was expected that programmers would want to add new methods to this hierarchy, the designers could have implemented the "Visitor" design pattern [Gamma et al. 1994]. However, as others have noted [Clifton et al. 2006; Millstein 2003], Visitor introduces its own problems: 1) its need must be anticipated in advance; 2) adding new classes to the hierarchy becomes difficult; 3) the visitXXX methods must all have the same return type, and may only throw unchecked exceptions (or only some particular checked exception); 4) inheritance among the classes to be visited can pose design issues.[3]

[3] To illustrate the last problem, suppose we have a visitor defined on the Reader class hierarchy. With the original visitor pattern [Gamma et al. 1994], the programmer would provide methods visitBufferedReader, visitStringReader, etc., but not visitReader. Unfortunately, this makes it impossible to put common functionality in a superclass case; each of the visitXXX methods must be overridden to perhaps call the same helper method. A more advanced variant of Visitor solves this problem [Vlissides 1999], but some frameworks and libraries still use the original pattern—even modern frameworks such as the Eclipse JDT.

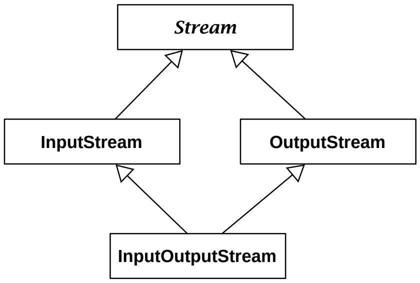

Figure 1.4: A stream class hierarchy forming an inheritance diamond

To solve these problems, I propose using external methods (described further below), which have been extensively studied in the literature (e.g., [Shalit 1997; Chambers 1992; Clifton et al. 2000; Allen et al. 2008]).

Reusing Code From Multiple Classes

While structural subtyping and external dispatch are useful features, they do not provide complete support for the kind of code reuse that is needed in practice [Ellis and Stroustrup 1990; Bracha and Cook 1990; Flatt et al. 1998; Ducasse et al. 2006]. For instance, in a single inheritance setting, there is no satisfactory solution when two or more classes need to share features that are not contained in their (unique) common parent. These features must either be pushed into the common parent (where it does not semantically belong) or they must be duplicated in the classes in question [Ducasse et al. 2006].

Consequently, various alternatives to single inheritance been proposed, such as multiple inheritance [Keene and Gerson 1989; Ellis and Stroustrup 1990; Meyer 1992], mixins [Bracha and Cook 1990; Flatt et al. 1998; Ancona et al. 2003], and traits [Schärli et al. 2003; Smith and Drossopoulou 2005; Flatt et al. 2006; Reppy and Turon 2007; Bergel et al. 2008]. Unfortunately, each of these designs has its own drawbacks. Multiple inheritance suffers from the diamond problem (described below), mixins must be applied linearly and may not inherit from one another, and traits may not contain state.

Diamond inheritance describes the situation when a class *C* inherits an ancestor *A* through more than one path (e.g., InputOutputStream's relationship to Stream in Fig. 1.4). This is particularly problematic when the class at the top of the diamond (e.g., Stream) has fields—should classes like InputOutputStream inherit multiple copies of the fields or just one? Virtual inheritance in C++ is designed as one solution for obtaining the latter semantics [Ellis and Stroustrup 1990]. But with only one copy of Stream's fields, object initializers are a problem: if InputOutputStream transitively calls Streams's constructor or initializer, how can we ensure that it is called only once? Existing solutions either restrict the form of constructor definitions [Odersky 2007] or ignore some constructor calls [Ellis and Stroustrup 1990].

There is another consequence of the diamond problem: it causes multiple inheritance to interact poorly with modular typechecking of external and multiple dispatch—my proposed solution to the problem of the previous subsection. In particular, in the presence of multiple inheritance, ambiguous external method definitions are difficult to detect in a modular manner.

To illustrate this problem, suppose we have the inheritance diamond of Fig. 1.4. Now, if we define external method m on both InputStream and OutputStream, there would be an ambiguity if m were called on an object of type InputOutputStream, as neither definition of m is more specific than the other. Detecting this situation would require either searching for subclasses of InputStream and OutputStream when m is defined (a non-modular check), or searching for external method definitions like m when InputOutputStream is defined (also non-modular).

Interestingly, this problem arises even with restricted forms of multiple inheritance, such as traits or Java multiple interface inheritance. Previous work either disallows multiple inheritance across module boundaries [Millstein and Chambers 2002], or burdens programmers by requiring that they always provide (possibly numerous) disambiguating methods [Frost and Millstein 2006; Allen et al. 2007].

1.3 Unity

I propose a new language, Unity, to solve the aforementioned problems. In this section I describe Unity at a high level and give an overview of how it can, in fact, solve these problems. Unity has three key features: structural subtyping, external dispatch, and multiple inheritance.

Structural Subtyping

Structural subtyping provides a solution to both the problem of retroactive abstraction and that of composing types. Structural subtyping is very popular in the research community, and has been extensively studied in a formal setting [Cardelli 1988; Bruce et al. 2003; Fisher and Reppy 1999; Leroy et al. 2004; Malayeri 2009a]. In a language with nominal subtyping (such as all mainstream statically-typed object-oriented languages), a type U is a subtype of T if and only if it is *declared* to be. In a language with structural subtyping, on the other hand, a type U is a subtype of T if its methods and fields are a superset of T's methods and fields (possibly with refined types). The interface of a class is simply its public fields and methods; there is no need to declare a separate interface type.

Structural subtyping offers a number of benefits, including the ability to create retroactive abstractions—new types that have a supertype relationship to existing types. In our example above, we would simply create a structural type Bitmap (with the setPosition and draw methods) and it would automatically be a supertype of both Drawable and Circle, without having to modify those types. This is illustrated in Fig. 1.5. In Unity, a *brand*[4] is similar to a class in Java-like languages. The type Bitmap is now used as the argument to centerAndDraw, with the result that either a Circle or an Icon may be passed to it.

This design also has benefits for code evolution: if a new method m with type τ is added to both Drawable and Circle, they are each subtypes of the structural type $\{m : \tau\}$ (and also Bitmap $\wedge \{m : \tau\}$). Finally, structural subtyping makes it trivial to compose types—the type Bitmap is automatically equivalent to the combination of the types { setPosition( ) } and { draw() }.

[4] The name "brand" is borrowed from Strongtalk [Bracha and Griswold 1993], which in turn borrowed it from Modula-3 [Nelson 1991].

```
type Drawable = Object (
    draw : unit → unit,
    setPosition : int * int → unit,
    setAlpha : int → unit
)

brand Circle extends Object (
    method draw ( ) : unit = . . .
    method setPosition ( x : int, y : int ) : unit = . . .
    method setAlpha ( alpha : int ) : unit = . . .
)

brand Icon extends Object (
    method draw ( ) : unit = . . .
    method setPosition ( x : int, y : int ) : unit = . . .
)

type Bitmap = Object (
    draw : unit→ unit,
    setPosition : int * int → unit,
)
```

```
let centerAndDraw = fn item : Bitmap -->
    . . .
    item.setPosition ( xpos, ypos ) ;
    item.draw ( )

. . .

centerAndDraw circle   // typechecks
centerAndDraw icon     // typechecks
```

Figure 1.5: Re-writing Fig. 1.1 using Unity's structural types. A brand declaration is similar to a class declaration in Java.

However, nominal subtyping has advantages as well, and a language that provides only structural subtyping would forgo these benefits [Pierce 2002; Ostermann 2008; Malayeri and Aldrich 2008a]. First, nominal subtyping allows the programmer to express and enforce design intent explicitly. A programmer's defined subtyping hierarchy serves as checked documentation that specifies how the various parts of a program are intended to work together. As a consequence, explicit specification has the advantage of preventing "accidental" subtyping relationships, such as the standard example of Cowboy.draw( ) and Circle.draw( ) [Magnusson 1991]. Nominal subtyping also allows recursive types to be easily and transparently defined, since recursion can simply go through the declared names. Third, error messages are usually much more comprehensible, since, for the most part, every type in a type error is one that the programmer has defined explicitly. Finally, as mentioned by Ostermann, nominal subtyping has a useful default of assigning blame to the definition of a type when a subtype relation does not hold. In contrast, structural subtyping defers blame to the point at subsumption is applied.

For these reasons, as well as to support external methods—which allow programmers to retroactively add new methods to existing classes—Unity provides *both* nominal and structural subtyping.[5] In the proposed design, a type has both a nominal and a structural component, and subtyping takes both components into account.

[5] See Sect. 2.2.3 for an explanation of why some form of nominal subtyping is extremely advantageous for supporting external dispatch.

```
String BufferedReader.readLine() { ... }    // look in buffer first
String StringReader.readLine() { ... }    // search string directly
String InputStreamReader.readLine() { ... }    // read an array of bytes
String Reader.readLine() { ... }    // call read() in a loop

Reader f = new FileReader("bar.txt");
String s = f.readLine();    // typechecks
```

Figure 1.6: Re-writing instanceof tests (Fig. 1.3) using MultiJava external methods

External Dispatch

External methods are like ordinary methods, but they can be added in a different module than the classes on which they perform dispatch.[6] External dispatch can make code more flexible and easier-to-evolve because the language no longer requires that the set of methods of a class be fixed when it is defined. External dispatch (and multimethod dispatch, a related feature) is supported by a number of languages, such as CLOS, Dylan, Cecil, MultiJava, and Fortress [Paepcke 1993; Shalit 1997; Chambers 1992; Clifton et al. 2000; Allen et al. 2008].

External methods solve the problem of adding new code that dispatches on an existing hierarchy. For instance, to add a new method to the Reader class, we simply write an external method readLine in a new module. Figure 1.6 shows such an external method written in MultiJava [Clifton et al. 2000]. This new method can be called just like an ordinary method; readLine can be called on any subclass of Reader, once the external method has been defined.

Unfortunately, while the new external method allows readLine to be called on Reader, it does not change the interfaces that Reader implements. Fig. 1.7 shows a MultiJava example where we would like to pass an object of type Reader to the method findString, which expects a first argument of type Readable. Even though we have added a new method to Reader to make it conform to Readable, without retroactive abstraction, the last line does not typecheck. Thus, the need for retroactive abstraction is even more apparent when new methods can be added to existing classes.

Fortunately, Unity provides a solution to this problem by including both structural subtyping and external dispatch, while retaining the modular typechecking of MultiJava. We could rewrite this example in Unity as in Fig. 1.8.

Here, the function findString takes as its first argument any object that has a readLine method that has only the receiver as an argument and that returns a string (i.e, type "() → string"). Once the external method readLine has been defined on Reader, all subtypes of Reader conform to this structural type and the last line typechecks.

[6] This document uses a very minimal definition of "module": a module is simply a set of definitions. The interface of the module is the types of those definitions.

```
interface Readable { String readLine(); }
boolean findString(Readable r, String s) { ... }
 // find string s in stream r

String Reader.readLine() { ... }    // new external method
f = new FileReader("bar.txt");
findString(f, foo);    // fails to typecheck!
```

Figure 1.7: External methods in MultiJava highlight the need for retroactive abstraction. The last line does not typecheck because Reader does not implement Readable, even though it has a readLine method.

```
// takes any object with a readLine method
let findString = fn (r : Object (readLine : () → string) , s : string) --> . . .

// external method definitions
method Reader.readLine : () ⇒ string = . . . // external method defined on ‘Reader’
method BufferedReader.readLine : () ⇒ string = . . .
method StringReader.readLine : () ⇒ string = . . .
. . .

using readLine in
    let f = new FileReader("bar.txt") in
    findString (f, "foo")  // typechecks!
```

Figure 1.8: Rewriting the code of Fig. 1.7 in Unity

Multiple Inheritance

As previously described, the root of the difficulty with multiple inheritance is the potential for inheritance diamonds; other issues with multiple inheritance (such as inheriting features with duplicate names) have been effectively solved by previous languages [Meyer 1992; Ellis and Stroustrup 1990].

Following this observation, Unity takes a novel approach: while permitting multiple inheritance, it disallows inheritance diamonds entirely. So that there is no loss of expressiveness, the notion of inheritance is divided into two concepts: an *inheritance dependency* (expressed using a requires clause, an extension of a Scala construct [Odersky and Zenger 2005; Odersky 2007]) and ordinary inheritance. Chapter 3 illustrates how programs that require diamond inheritance can be translated to a hierarchy that uses a combination of requires and multiple inheritance, without the presence of diamonds. As a result, Unity retains the expressiveness of diamond inheritance while avoiding its problems.

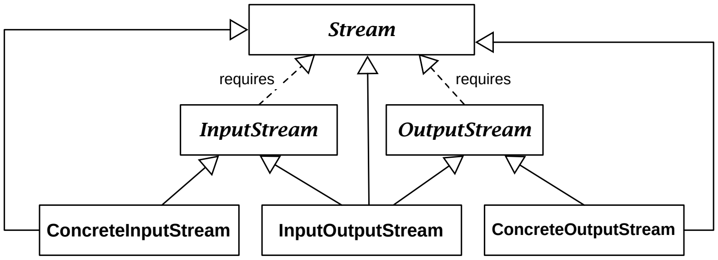

Figure 1.9: The stream diamond of Fig. 1.4 re-written in Unity. The requires relationship provides subtyping without inheritance.

To provide a sense of this translation, Fig. 1.9 shows how the inheritance diamond of Fig. 1.4 is translated to Unity. Essentially, inheritance diamonds are converted to *subtyping* diamonds—which are allowed by the language—using the requires construct. The details of the multiple inheritance design, as well as a discussion of the revised stream hierarchy, are described in Chapter 3.

1.4 Statement of the Thesis

The thesis of this dissertation is:

> *An object-oriented programming language can provide integrated support for (a) external dispatch, (b) nominal subtyping, (c) structural subtyping, and (d) multiple inheritance—all without sacrificing modular typechecking. These richer structuring mechanisms can serve to make code more reusable and adaptable.*

The thesis is affirmed through several hypothesis; each is described below along with a description of its supporting evidence.

Hypothesis I

A language with synergy between structural subtyping and external dispatch can be achieved through a novel combination of structural and nominal subtyping.

Validation:

- Unity language design and type system (Section 2.5).
- Type safety proof for the core language (Section 3.7.4 and Appendix A).

Hypothesis II

By providing retroactive abstraction, structural subtyping can be used to improve the reusability and maintainability of existing object-oriented programs.

Validation:

- Quantitative and qualitative analyses of open-source Java programs chosen from a variety of domains (Chapter 4):
    1. Evidence suggesting that structural subtyping could help make method parameters more general (Sect. 4.3).
    2. High frequency of *common methods*—methods with the same name and signature, but that are not contained in a common supertype of the enclosing classes (Sect 4.5.1).
    3. Low frequency of common methods that represent an accidental name clash (Sect 4.5.2).
    4. Evidence that some cases of code duplication could be avoided with structural subtyping (Sect. 4.5.3).

Hypothesis III

Existing language designs can lead to coding patterns that defer errors to runtime; structural subtyping could provide more static typechecking in these situations by allowing programmers to encode more properties directly in the type system.

Validation:

- Quantitative and qualitative data showing that:
    1. Some Java runtime exceptions (i.e., OperationUnsupportedException) can be eliminated in a straightforward manner with a design that uses structural subtyping (Sect. 4.4).
    2. Some uses of Java reflection can be converted to uses of structural subtyping (Sect. 4.7).

Hypothesis IV

The combination of structural subtyping and external dispatch has the synergistic effect of providing an expressive form of retroactive abstraction.

Validation:

- Examples illustrating the increase in expressiveness when these features are combined (Sect. 2.2).
- Results from empirical study showing that many cases of cascading instanceof tests in Java programs may be re-written using a combination of structural subtyping and external methods (Sect. 4.6), thereby allowing an existing class to be adapted to a new context.

Hypothesis V

Through the use of a novel multiple inheritance scheme, modular typechecking can be performed in a language with multiple inheritance and external dispatch, without requiring programmer-specified disambiguating methods.

Validation:

- Type safety proof for the core language (Section 3.7.4 and Appendix A).
- Detailed argument describing modularity of Unity type system (Section 3.7.3).

Hypothesis VI
A language can be designed with a new form of multiple inheritance—multiple inheritance without diamonds—a design that provides more opportunities for code reuse and that is more expressive than other proposed alternatives to full multiple inheritance (i.e., multiple interface inheritance, mixins, and traits).

Validation:

- Unity multiple inheritance design (Chapter 3).
- Detailed comparison to the existing proposals in the context of an example in Unity (Sect. 3.5).

Hypothesis VII
By converting inheritance diamonds to inheritance dependencies and subtyping among abstract classes (via a requires clause), a program with inheritance diamonds can be systematically translated into a program with multiple inheritance but without any inheritance diamonds.

Validation:

- Real-world examples showing how C++ inheritance diamonds can be systematically translated to Unity (Section 3.6).

Conventions

This document makes use of the following typographical conventions:

- `monospace` is used for Java code listings
- (proportional-width) sans serif[7] is used for Unity code listings, all inline code references, and (in the formal system) keywords
- *utopia italic* is used for metavariables (e.g., B, C, D) and auxiliary functions (e.g., *mtype*)
- Small Caps is used to name inference rules
- **boldface** is used to name judgements (e.g., p **ok**)

[7] It is more visually pleasing than `monospace`.

Chapter 2

Structural Subtyping and External Dispatch

> "Must a name mean something?" Alice asked doubtfully.
> "Of course it must," Humpty Dumpty said with a short laugh:
> "my name means the shape I am—and a good handsome shape it is,
> too. With a name like yours, you might be any shape, almost."
>
> Lewis Carroll (*Through the Looking-Glass*)

> If it looks like a duck, and quacks like a duck, we have at least to consider the possibility that we have a small aquatic bird of the family Anatidae on our hands.
>
> Douglas Adams (*Dirk Gently's Holistic Detective Agency*)

This chapter describes one of the main contributions of Unity: the combination of structural subtyping and external dispatch.[1] A clean integration of these features is achieved using a combination of nominal and structural subtyping. The chapter introduces Unity through a series of examples, describes practical applications of the work (design patterns and optional methods), and presents the full formalization of the Unity calculus. The multiple inheritance aspects of the calculus are visually indicated and will be described in Chapter 3.

2.1 Overview of Unity

In Unity, an object type is a value (usually a record) tagged with a *brand*. Brands induce the nominal subtyping relation, which I call "sub-branding." Brands are nominal in that the user defines the sub-brand relationship, like the subclass relation in languages like Java, Eiffel, and C++.

[1] The main contributions of this chapter appeared in previous publications [Malayeri and Aldrich 2007; 2008a].

The structural subtyping component of the system is used for subtyping the set of methods of a brand (denoted by $\{m_i : \tau_i \ ^{i \in 1..n}\}$). The usual structural depth and width subtyping rules apply to this method set. If B has methods $m : \tau$ and $n : \sigma$, then an object o with brand B conforms to the types $B(m : \tau, n : \sigma)$, $B(m : \tau)$, $B()$, and $B(n : \sigma')$ (where σ is a subtype of σ').[2] Also, since Object is the root of the inheritance hierarchy, the object o also has type Object$(m : \tau, n : \sigma)$. This last relation is achieved through the combination of nominal and structural subtyping.

In particular, the system has the following rule for general subtyping (denoted by "≤"):

Definition 2.1 (General subtyping).
$B(m_i : \tau_i \ ^{i \in 1..n}) \leq C(n_j : \sigma_j \ ^{j \in 1..x})$ if and only if:

- B is a sub-brand of C, and
- $\{m_i : \tau_i \ ^{i \in 1..n}\}$ is a structural subtype of $\{n_j : \sigma_j \ ^{j \in 1..x}\}$

In turn, structural subtyping obeys the following rule:

Definition 2.2 (Structural subtyping).
$\{m_i : \tau_i \ ^{i \in 1..n}\}$ is a structural subtype of $\{n_j : \sigma_j \ ^{j \in 1..x}\}$ if and only if:

- $\{n_j \ ^{j \in 1..x}\} \subseteq \{m_i \ ^{i \in 1..n}\}$, and
- $m_i = n_j$ implies $\tau_i \leq \sigma_j$

In other words, the set of labels in the second structural type N must be a subset of the set of labels in the first type M, and for identical labels m_i and n_j, the corresponding types must be in the (general) subtype relation.

Unity has two kinds of method declarations: internal and external.[3] Internal methods are defined within a brand declaration, similar to methods in Java-like languages. External methods in Unity are similar to those in MultiJava and related languages [Clifton et al. 2006; Millstein 2003]; they may be defined outside of a brand but perform dispatch and may be overridden.[4]

To support information hiding, MultiJava does not permit an external method defined on class C to access C's private members. Similarly, in Unity, external methods may not access any of the corresponding brand's fields. Sub-brands, however, may access super-brand fields, so fields in Unity are like C++ "protected" members.

[2] This is not strictly the case in the formal system, which distinguishes between simple and qualified method names. Essentially, qualified method names are not included an object's structural type, but may be called using nominal method lookup. Structural method names are added to an object explicitly using mapping expression (which would not appear in the surface syntax). However, the examples that follow assume that all of the methods of a brand appear in the structural type of its objects and do not make a distinction between simple and qualified method names.

[3] To simplify the discussion, the remainder of this document abbreviates "internal method declaration" as "internal method," and analogously for "external method declaration."

[4] This is in contrast to "extension methods" in C# 3.0, which are merely syntactic sugar for static methods defined in a helper class.

Note that external methods are closely related to multimethods, which are methods that may dispatch on *any* subset of their arguments—not just the receiver. The formal system does not include multimethods however, as there is a straightforward modular encoding of asymmetric multimethods using external methods.[5] Consequently, the same typechecking issues apply to both external methods and asymmetric multimethods.[6]

2.2 Unity by Example

The section presents, by example, the intuition behind Unity and situations in which it can be useful. The examples also demonstrate the synergy between structural subtyping and external methods. I also provide a detailed comparison of Unity to other related designs, in the context of the examples.

2.2.1 Example 1: Streams

The first example involves the use and implementation of character-based input and output streams.

Defining a type. For input streams, we first wish to create a "Reader" abstraction, which represents any object that has read, skip, and close methods (with appropriate types).

To define this abstraction, we use a type declaration (Fig. 2.1), which defines a *type abbreviation*. The nominal component (i.e., the brand) of Reader is Object and its structural component consists of the methods read, skip and close, along with their types.

As in Java-like languages, the brand Object is the root of the inheritance hierarchy; all brands directly or indirectly extend Object. Note that the code listings omit the extends clause for direct sub-brands of Object (e.g., AbstractReader).

Reader contains two different arrow types, "⇒" and "→." The first, "⇒," is used for method types. To the left of this arrow is the receiver's *structural* type. The nominal component is omitted—it can always be inferred from context in which the method type appears. For example, Object is the nominal component of the method read in the type abbreviation Reader.

To simplify the formal system, methods take only one argument: the receiver (i.e., this). If additional arguments are needed, ordinary first-class functions are used; these have types containing the "→" arrow. Functions are defined using the "fn x: t --> e" syntax.

For example, skip has type "( ) ⇒ long → long." Since the receiver's structural type is empty, this specifies that skip can be applied to any object with the appropriate (implicit) nominal type

[5] With *asymmetric* multimethods, the order of arguments affects dispatch, in contrast to symmetric dispatch. An asymmetric multimethod dispatching on brands $B_1, \ldots, B_n$ can be translated to external methods defined on each B_i, where each method calls the method in brand B_{i+1}, with the actual code defined in the method on B_n.

[6] Multimethods with symmetric dispatch semantics introduce a few orthogonal typechecking issues; see [Millstein and Chambers 2002; Clifton et al. 2006; Millstein et al. 2004; Millstein 2003] for a more detailed discussion.

```
type Reader = Object (read: ( ) ⇒ char,
                      skip: ( ) ⇒ long → long,
                      close: ( ) ⇒ unit)  // all methods take reciever of type Object( )

brand AbstractReader ( . . . ) // default implementation for methods in Reader

brand CharArrayReader extends AbstractReader (
    array: char[ ]; // field
    method read = . . .; method skip = . . .; method close = . . .  // same types as in Reader
    method mark: ( ) ⇒ unit = . . . // save current position
    method reset: ( ) ⇒ unit = . . . // reset to position saved by 'mark'
    method seek: ( ) ⇒ long → long = ( fn pos: long --> . . . )  // seek to 'pos'
)

brand BufferedReader extends AbstractReader (
    method read = . . .; method skip = . . .; method close = . . .  // same types as in Reader
    method mark: ( ) ⇒ unit = . . .
    method reset: ( ) ⇒ unit = . . .
)

// does not extend AbstractReader
brand SomeOtherReader ( . . . ) // implementation of all of the Reader methods

// define function that finds string 's' in the 'r' stream
let findString = fn (r: Reader) (s: string) : long --> . . .

// all typecheck; each is a subtype of Reader
findString bufReader "foo"
findString charArrayReader "bar"
findString someOtherReader "baz"
```

Figure 2.1: Stream example illustrating brand extension and structural types

(here, Object). Since skip needs a additional argument (the number of bytes to skip), its return type is a function that takes a long and returns a long.

In essence, the nominal type of the receiver is not specified in method types, as a method type always appears with a surrounding brand. Therefore, in the type $B(m : (\,) \Rightarrow \tau)$, m's receiver has type $B(\,)$ (i.e., B with no additional structural constraints).[7]

[7] It would be possible to include the receiver in method types, but then the formal system would then have to ensure consistency between receiver types and their enclosing brand type. This would serve only to add unnecessary complexity.

```
// external methods: add methods to AbstractReader and CharArrayReader,
// with (static) structural constraint that receiver must have 'mark' and 'reset' methods
method AbstractReader.parse: ( mark: ( ) ⇒ unit, reset: ( ) ⇒ unit ) ⇒ unit =
    // 'this' has type AbstractReader(mark: . . ., reset: . . .)
    Ⓐ . . . // parse text using 'mark' and 'reset'

method CharArrayReader.parse: ( ) ⇒ unit =
    Ⓑ . . . // more efficient parsing using method 'seek'
```

Figure 2.2: Defining an external method (with structural constraints) on AbstractReader and CharArrayReader

Defining brands. In Fig. 2.1, I have also defined a "reader" brand, AbstractReader, which contains default implementations of the Reader methods. The brands CharArrayReader and BufferedReader each extend AbstractReader and provide additional functionality. We may also define reader brands that do not extend AbstractReader but still conform to the Reader type; the brand SomeOtherReader is an example.

Now, if we write a function findString that operates on objects of type Reader, it may be used on any of the brands we have defined, as they all conform to the Reader type. This is illustrated by the last three lines of the code listing.

External methods. Thus far, we have used only internal methods and ordinary functions. Since Unity also provides external methods, we can define new functionality in a new module. That is, methods do not need to appear in the same module as that of the brands on which they operate; Fig. 2.2 contains such a definition. Here, I have defined an external method parse with two implementations—for each of AbstractReader and CharArrayReader. This second override could perhaps perform more efficient parsing using the seek method in CharArrayReader.

The external method defined on AbstractReader also defines an additional *structural constraint* on the receiver: it must contain the mark and reset methods, in addition to having brand AbstractReader. Consequently, parse may only be called on objects that conform to the type AbstractReader(mark: ..., reset: ...). Note that the structural constraint is not needed for CharArrayReader, since all objects of this type already have methods mark and reset.

Internal methods may also specify structural constraints; Sect. 2.2.3 below describes how this can be used to encode Java-style abstract methods.

Note that the structural constraint is purely a static concept; there is no structural method dispatch (described further in Sect. 2.2.3). That is, parse may only be called by objects that are *statically* known to contain mark and reset methods. Consequently, it is a type error to define two methods that differ only in their structural constraints.

External methods can be overridden by internal methods in sub-brands. For example, suppose we wish to define a type of reader that natively supports parse functionality. This is cap-

```
brand ParseableReader extends BufferedReader (
    . . . // implements all BufferedReader methods

    // overrides external method parse in Fig. 2.2
    method parse: ( ) ⇒ unit = Ⓒ  . . .  // specialized parse algorithm for this type of stream
)

brand SimpleReader extends AbstractReader( . . . ) // contains only AbstractReader methods

using parse in
    bufReader.parse   // typechecks; BufferedReader has methods 'mark' and 'reset'. calls code (A)
    charArrayReader.parse   // typechecks. calls code (B)
    simpleReader.parse  // doesn't typecheck: need methods 'mark' and 'reset'
    parseableReader.parse // calls code (C) above

    let absReader: AbstractReader(parse: ( ) ⇒ unit) =  parseableReader in
        absReader.parse  // typechecks, calls code (C) above

// doesn't need to be in using block, has its own implementation
parseableReader.parse // calls code (C) above
```

Figure 2.3: Overriding and using the external method parse

tured by the brand ParseableReader in Fig. 2.3. Assuming the external method definition of Fig. 2.2 is in scope, ParseableReader may provide its own version of parse.

However, to call the external method, it must be imported into a lexical scope via the using expression, as illustrated at the end of the code listing. An external method behaves just as an ordinary method; dynamic dispatch occurs for the expressions "bufReader.parse" and "charArrayReader.parse." However, since SimpleReader does not contain the mark and reset methods, the expression "simpleReader.parse" does not typecheck. (Below, we will see how external methods can be used to make a brand conform to a particular structural constraint.) Note that parse may be called on any ParseableReader object without it being in the using block, as the override of AbstractReader.parse implicitly imports the method definition for objects of type ParseableReader.

Adding "write" functionality. Next, we add code for writing to a character stream. The type Writer represents an object that supports basic write functionality; AbstractWriter provides default implementations for these methods (Fig. 2.4). We define the brand StringStream, which allows both reading and writing. Note however, without multiple inheritance (which will be introduced in Chapter 3), StringStream may only extend *one* of AbstractWriter and AbstractReader (this second type is commented out in the extends clause).

```
type Writer = Object (append: ( ) ⇒ char → Writer,
                      write: ( ) ⇒ int → unit,
                      flush: ( ) ⇒ unit,
                      close: ( ) ⇒ unit)

brand AbstractWriter (. . .) // default implementation of Writer methods

brand StringStream extends AbstractWriter //, AbstractReader [no multiple inheritance yet]
    method read = . . . ; method skip = . . . ; method close = . . . // same types as in Reader
    method append = . . . ; method write = . . . ; method flush = . . . // same types as in Writer
    method seek: ( ) ⇒ long → long = . . .
)

// may use all methods in Reader, Writer and also seek
let readAndWrite = fn (stream:  Reader ∧ Writer ∧ seek: ( ) ⇒ long → long) -->
    s.read ; s.seek 10
    s.write ‘f’ ; s.flush

readAndWrite stringStream // typechecks
```

Figure 2.4: Adding “writer” definitions

Using intersection types (with the “∧” notation), we can easily combine type definitions. The parameter to readAndWrite, for example, must be a subtype of both Reader and Writer and also have a seek method. Since StringStream conforms to this type, it may be passed as a parameter to readAndWrite.

Adding code to satisfy a structural constraint. I have described how function and method arguments, including the receiver of internal and external methods, may specify structural constraints—methods that must exist in addition to the brand’s defined methods. If a brand does not conform to this structural constraint, external methods can be used to remedy this situation.

Figure 2.5 shows such an example. Here, I have defined the writeLine external method, which is applicable to any AbstractWriter that provides a newline method. This latter method is expected to return the newline string for the current platform.

Since AbstractWriter does not define newline, the expression writer.writeLine does not typecheck. But, once we have added newline as an external method and imported it into the current scope (with a using expression), writeLine may be called on any object of type AbstractWriter.

Figure 2.6 shows the subtyping relationships that hold as a result of the brand and external method definitions. These relationships illustrate why the example code typechecks properly.

```
// external method: add method to AbstractWriter,
// with constraint that receiver must have a 'newline' method
method AbstractWriter.writeLine: (newline: ( ) ⇒ string) ⇒ string → unit =
    fn s: string --> . . . // print out 's,' using newLine method

new StringStream( ).writeLine "42" // doesn't typecheck—no 'newline' method in StringStream

method AbstractWriter.newline: ( ) ⇒ string = "\r\n"

using newline in
    new StringStream( ).writeLine "42"  // typechecks!
```

Figure 2.5: Adding external methods to conform to structural constraints

2.2.2 Example 2: Collections

The second example regards the definition and implementation of "collection" classes (e.g., a map, set, etc.). This section illustrates how abstract methods can be encoded via structural constraints and how nominal types can be used to enforce design intent.

Encoding abstract methods. Using structural constraints on a method's receiver, we can effectively encode abstract methods. The first benefit of this encoding is that it simplifies the formal system; we need not include abstract methods or abstract classes.

To illustrate this encoding, consider the Java class definition at the top of Fig. 2.7, which is based on a Java 1.5 Collections Library class.[8]

To translate this code to Unity, the abstract methods in the Java class are converted to structural constraints on the receiver of the relevant methods (bottom part of Fig. 2.7). For instance, contains is implemented in terms of iterator, so it requires that its receiver have the latter method; the same pattern is used for isEmpty and toString.

Not only does this encoding simplify the formal system, it increases the language's flexibilty; a structural constraint can be satisfied using either an internal or external method. Due to a restriction on the definition of external methods, if iterator and size are introduced as abstract internal methods, subclasses can only provide *internal* method implementations. With a structural constraint, however, one of a set of external methods definitions may be chosen to satisfy it.

In particular, the restriction is that external methods may not override internal methods (described further in Sect. 2.4.2). As a consequence, once a method is defined as an "abstract" internal method, it can only be implemented with internal methods in sub-brands.

With structural constraints, on the other hand, in some module M_1, we may define:

[8] Note that not I am not using parametric polymorphism here; an extension of Unity with this feature is outlined in Sect. 2.5.3 and formalized in [Malayeri and Aldrich 2008b].

AbstractReader ≤ Reader
CharArrayReader ≤ AbstractReader
BufferedReader ≤ AbstractReader
SomeOtherReader ≰ AbstractReader
SomeOtherReader ≤ Reader
CharArrayReader ≤ AbstractReader(mark: …, reset: …)

StringStream ≤ Reader
StringStream ≤ Writer

Writer ∧ seek ≤ Writer
StringStream ≤ Writer ∧ Reader ∧ seek

// after a 'using' expression
new StringStream( ): StringStream(newline: …) ≤ AbstractWriter(newline: …)

Figure 2.6: Typing and subtyping induced by the brand declarations. Types of methods are elided, and empty structural components (i.e., ( )) are omited.

```
// ordinary iterator implementation
method iterator AbstractList( ): Iterator = . . .
method iterator StoreBackedList( ): Iterator = . . .
```

Now, suppose that in another module M_2, we have a different implementation for iterator (perhaps one that caches the next element to be retrieved). Then, depending on which external method is imported through the "using" expression (M_1.iterator or M_2.iterator), different external method definitions can be "plugged-in" to a particular context.[9]

This feature essentially allows different external methods to be attached to existing objects. This is reminiscent of mixins [Bracha and Cook 1990; Ancona and Zucca 1996], but here the "mix-in" operation occurs at the object level. Sections 2.4.3 and 2.5 describe the formalization of the "using" construct.

Combining structural and nominal types. Up to this point, we used brands to implement (internal and external) methods. But, it is also possible to use brands to specify and enforce design intent.

In particular, let us consider the types Collection and Set from the Java Collections library. These types have identical interfaces, but are not necessarily used in the same way. In particular, a Set does not have duplicate elements, while a Collection may.

Unity's nominal typing component—brands—can be used to enforce the intent that an objects should have a particular inheritance path, in addition to having methods with the appro-

[9] To simplify the presentation, the "using" expression the code examples did not specify a module, but this would be a straightforward extension.

```
// Java class with abstract methods
abstract class AbstractCollection {
   abstract Iterator iterator();
   abstract int size();
   boolean contains(Object o) { ... } // uses iterator
   boolean isEmpty() { ... } // uses size
   String toString() { ... } // uses iterator
}

// Unity translation
type Iterator = Object(next: () ⇒ Object(), hasNext: () ⇒ bool)

brand AbstractCollection (
      method contains: (iterator: () ⇒ Iterator) ⇒ Object() → bool = . . .
      method isEmpty: (size: () ⇒ int) ⇒ bool = . . .
      method toString: (iterator: () ⇒ Iterator) ⇒ string = . . .
)
```

Figure 2.7: Translating Java abstract methods to Unity structural constraints

```
// same method types as in AbstractCollection
type Collection = Object(contains: . . . , isEmpty: . . . , toString: . . . , iterator: . . ., size: . . . )

brand SetBrand () // define a brand to distinguish Set from Collection
type Set = SetBrand(contains, isEmpty, toString, iterator, size) // same types as in Collection

// a concrete set implementation
brand HashSet extends SetBrand ( . . . ) // declare and implement Set methods

type Map = Object (
      entrySet: () ⇒ Set
      values: () ⇒ Collection
      . . . )

let useMap = fn (m: Map) --> let set = m.entrySet in . . . // ‘set’ has brand SetBrand
```

Figure 2.8: Using nominal types to create constraints

```
type Readable = Object( read: ( ) ⇒ char )
type Writeable = Object( write: ( ) ⇒ char → int )

method Readable.foo = . . .  // (1)
method Writeable.foo = . . .  // (2)

readWritable.foo  // ambiguous method call!
```

Figure 2.9: In Unity, dispatch is performed on brands, since structural method dispatch would result in ambiguous method calls.

priate types. Figure 2.8 has an example of this. Here, in the useMap function, we know that the variable set has brand SetBrand. Since an object cannot implicitly conform to this type (as its brand must be a sub-brand of SetBrand), this can help prevent "accidental" subtyping.

Note that SetBrand has not defined any methods as it is conceptually an interface; any included methods would be "abstract." For this reason, the relevant methods have instead been moved to the type Set.

I refer back to this code listing in Section 2.6 below, in the context of a comparison to "where" clauses in Cecil.

2.2.3 Discussion and Summary

I this section, I discuss issues surrounding Unity's dispatch semantics and summarize the key features of the language.

Dispatch Semantics

In Unity, external dispatch may only be performed on brands; this restriction is necessary to make ambiguity checking feasible. As a counter-example, suppose we *were* to allow dispatch on structural types. If structural types Readable and Writeable were defined as in Fig. 2.9, we could write a method foo that behaves differently depending on whether its receiver conforms to the Readable type or the Writeable type (an admittedly contrived, though illustrative, method). Aside from making it difficult to efficiently implement method dispatch (in the worst case, the entire structure of the type would have to be examined at runtime), this definition is ambiguous: what if foo is called on an object that is a subtype of both Readable and Writeable?

If dispatch were permitted on structural types, these kinds of ambiguities would continually arise, due to the intrinsic properties of structural subtyping. To ensure type safety in such a case, the typechecker would require that the programmer provide disambiguating methods whenever there is any potential ambiguity (based on the static structure of the program); this is the only way to statically prevent all runtime ambiguities in a modular manner.[10] However, this approach

[10] I am, of course, assuming that it is unacceptable for the runtime to arbitrarily choose one of the candidate methods, or to choose a method based on the textual ordering of the definitions.

is infeasible: for a particular external method, the number of required disambiguating methods is exponential in the number of implementations dispatching on incomparable types (i.e., types where neither is a subtype of the other). The need for disambiguating methods is discussed further in Sect. 3.3, in the context of "diamond" multiple inheritance.

This design has another issue: it can magnify the problem of accidental subtyping. Suppose we were to have the following declaration:

method Object.bar(x: τ_1) = . . . Ⓐ *// called when receiver dynamically has method 'x'*
method Object.bar(x: τ_1, y: τ_2) = . . . Ⓑ *// called when receiver dynamically has methods 'x' and 'y'*

This definition is not ambiguous (in the sense of the previous example), but it can have an unexpected interaction with external methods. In particular, suppose an external method y (with the appropriate type) is added to an object o of brand C, which previously had only an x method. Now, calls to o.bar will change—code fragment (B) will be executed! This is the same sort of problem as with accidental subtyping, but is even more subtle and difficult to statically diagnose.

Summary

The examples illustrated the three main features in Unity: structural types, nominal types, and external dispatch:

- **Structural types** can be used to create *structural constraints*. If method m has structural constraints $(m_1 : \tau_1, \ldots, m_n : \tau_n)$, the expression $o.m$ is valid only if o's structural type contains $m_1, \ldots, m_n$ with types conforming to (i.e., subtypes of) $\tau_1, \ldots, \tau_n$.[11]
- **Nominal types** are used to create a new brand that can be used in dispatch; as a consequence, programs can *define new behavior* for the newly defined brand. In the streams example, CharArrayReader is defined as an extension of AbstractReader because (for example) the behavior of parse is different for each type.

  The programmer can also use brands to preserve design intent; nominal types can be used to distinguish between similarly-named methods that behave differently (e.g., Cowboy.draw( ) and Circle.draw( )). This was shown in the collections example, with SetBrand.
- **External dispatch** and structural subtyping have synergistic properties. Structural subtyping can be used to *specify the constraints* of a method, and external methods can be used to make existing brands *conform* to those constraints. Additionally, the "using" expression, in conjunction with external methods, can be used to "plug-in" different internal method implementations to different contexts.

[11] This is an over-generalization for the purposes of presentation. As we will see in the formal system (Sect. 2.5), the expression $o.m$ is also valid if o's *brand* contains method m.

2.3 Structural Subtyping and Design Patterns

In this section, I consider two potential applications of Unity: expressing three commonly-used design patterns, and redesigning the Java Collections Library to remove "optional" methods.

2.3.1 GoF Patterns in Unity

Several design patterns identified by [Gamma et al. 1994] can be expressed more elegantly in Unity versus mainstream object-oriented languages. These include Visitor, Proxy, and Decorator.

The need for Visitor is in fact obviated in Unity, due to the presence of external methods. To dispatch on an existing hierarchy, the programmer simply writes new external methods for those brands [Clifton et al. 2006; Millstein 2003].

In some situations, the Proxy and Decorator patterns are easier to implement in a language with structural subtyping. In order to effectively use these patterns in a nominally-typed language, interfaces must be defined in advance for the classes for which we wish to define a proxy or decorator. If no appropriate interface exists, these patterns may be impossible or unwieldy to implement.

Proxy. The Proxy design pattern is used to create an intermediary to an object to be used in the object's place. The object in question may be expensive to create or should perhaps be accessed in a particular manner. Examples include reference-counted pointers and local objects that access remote resources.

Suppose we wish to create a proxy for the RealSubject class, which implements the Subject interface. The typical implementation is to create a Proxy class that also implements Subject and has a field of type RealSubject. Proxy can then forward method calls to the RealSubject field, possibly performing additional operations before and after the method call.

This implementation depends on the existence of the Subject interface; without such an interface, problems will arise. Concretely, suppose we have a class RasterImage that does not implement any interfaces. If we were to use a traditional nominally-typed language, the new RasterImageProxy class would have to extend RasterImage and re-implement the necessary methods. Inheritance would be necessary here, in order to obtain the appropriate subtyping relation. But, there is a problem if RasterImage loads the image into memory in the constructor, since the RasterImageProxy constructor must call the RasterImage constructor. This would then make it impossible for RasterImageProxy, for example, to perform lazy loading of the image file.

This problem does not occur with a language that separates inheritance and subtyping, like Unity. In such languages, well-designed code would never mention the nominal type RasterImage directly (except when creating an object), and would instead use a type whose structural component consisted of the methods of RasterImage. (Analogously, in a nominally-typed language that separates inheritance and subtyping, the *type*, rather than the class, corresponding to RasterImage would be used.) Then RasterImageProxy need not inherit from RasterImage, but would instead have a field of type RasterImage—the usual implementation of the Proxy design pattern.

Decorator. A similar situation arises with the Decorator design pattern. Decorator is often used when we wish to add new or additional behavior to an existing class. The pattern is implemented by creating a new class that "wraps" the original class and forwards method calls to it. To create a decorator for a ConcreteComponent class that implements the IComponent interface, we would create a class Decorator containing a field of type IComponent. Similar to proxy, the Decorator class can forward calls to this field, possibly performing additional behavior before or after the method call. (Note that though the implementation of Proxy and Decorator is similar, the decorated object often changes dynamically, whereas a proxy does not typically change its subject object.) Decorators are often used to attach new behavior to GUI objects, such as adding a scrollbar or border. The Java stream library also uses decorators to implement functionality such as buffering and encryption.

As with Proxy, traditional languages can make it difficult to implement Decorator when the ConcreteComponent class does not implement a suitable interface. The workaround is to create a class Decorator that inherits from ConcreteComponent (in order to obtain subtyping), and *also* defines a ConcreteComponent field. But, this design will only work if ConcreteComponent has not been declared as final. Additionally, in this awkward design, unused resources may be created. Programmers must also take care to never call methods in ConcreteComponent on the this object, but rather on the declared ConcreteComponent field.

Again, if inheritance and subtyping are distinct features, client code would use the type ConcreteComponent (rather than the class), and Decorator would declare itself a subtype (but not a subclass) of ConcreteComponent. With structural subtyping, the code is even simpler, as Decorator need not declare a relationship to ConcreteComponent.

Concretely, the code for a typical Java Decorator implementation is:

```
interface IComponent { void doSomething(); }
class ConcreteComponent implements IComponent { void doSomething() { ... } }

class Decorator implements IComponent {
   IComponent wrapped;

   void doSomething() {
      ... // set−up code
      wrapped.doSomething();
      ... // tear−down code
   }
}
```

The corresponding code in Unity would be:

```
brand ConcreteComponent (method doSomething(): unit = . . . )

brand Decorator (
    wrapped: Object(doSomething: () ⇒ unit);
    method doSomething(): unit =
        . . . // set−up code
        wrapped.doSomething
        . . . // tear−down code
)
```

2.3.2 Optional Methods in the Java Collections Library

In this section, I describe the tradeoffs that a library designer must make when using a language that has only nominal subtyping. The design of the Java collections library illustrates that designers would rather circumvent the type system than have a proliferation of types. I believe this situation can occur all too often in a language with only nominal subtyping. Chapter 4 presents empirical evidence to support this claim.

In the Java collections library, the interface java.util.Collection has several "optional" methods: add, addAll, clear, remove, removeAll, and retainAll. Many of the abstract classes implementing Collection (e.g., AbstractCollection, AbstractList, AbstractSet) throw an UnsupportedOperationException when those methods are called. There are a total of 30 optional methods in java.util.*, and java.lang.Iterator has an additional optional method. The methods were designed this way to avoid an explosion of interfaces such as MutableCollection, ImmutableCollection, etc., and a corresponding increase in the number of sub-interfaces (e.g., MutableList, ImmutableList, etc.) [Sun Microsystems 2003].

Let us consider a Java collections framework without the optional methods. Figure 2.10 shows a relevant portion of the current Java collections hierarchy. Figure 2.11 show refactored AbstractList and AbstractSet classes with finer grain behavior, with new interfaces that capture the distinction of modifiability directly in the hierarchy—doing away with optional operations. The portions of AbstractList that pertained to mutability have been moved to AbstractModifiableList; the same for AbstractModifiableSet. The interface Collection<E> represents a collection that is possibly-modifiable, while ModifiableCollection<E> represents a collection that can be modified. Accordingly, its iterator() method returns a ModifiableIterator. This new Iterator interface is depicted in Figure 2.12. The Iterator<E> interface has been changed so that it no longer has a remove() operation; this method has been moved to ModifiableIterator<E>. There are now two new ListIterator interfaces, one for fixed-size lists, and one for variable-size lists. These correspond to the ModifiableFixedSizeList<E> and ModifiableList<E> interfaces in Figure 2.11. The hierarchy for Set is similar to that of List (though simpler, since there are no fixed-size sets, and no set-specific iterator).

Figure 2.13 shows the refactored Map interface and related classes. The main interface here is Map<K,V> which has a method entrySet(). In the original collections hierarchy, this returns a Set<Map.Entry>, but the documentation states that the returned set supports only set removal operations, not set addition operations. So, an additional interface is needed for a set that sup-

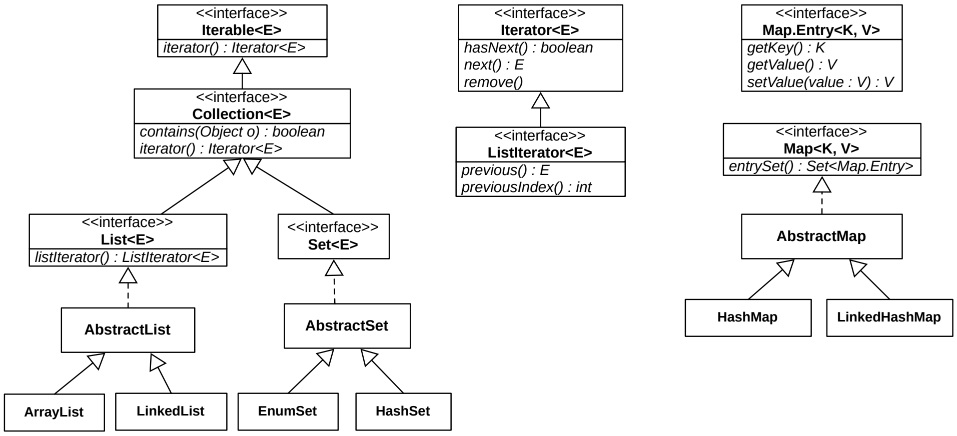

Figure 2.10: A portion of the Java collections framework. Only a subset of an interface's methods are listed. Type parameters are elided in classes.

ports modification only through remove operations; this is represented by RemovableSet in Figure 2.14. Another interface is needed for a general collection (as opposed to a set) that supports element removal, since the method values( ) returns an object of such a type. This is represented by RemoveableCollection (also in Figure 2.14).

As noted, in the original design entrySet( ) returns type Set<Map.Entry>. This translates into four possibilities in the refactored hierarchy: Set<Map.Entry> (a read-only set with read-only entries), RemovableSet<Map.Entry> (a mutable set with read-only entries), Set<ModifiableMap.Entry> (a read-only set with mutable entries), and RemovableSet<ModifiableMap.Entry> (a mutable set with mutable entries). This is due to the fact that Map.Entry.setValue is an optional method, and thus needs a new interface to capture its behavior. Aside from this proliferation of interfaces, the class diagram for Map is fairly straightforward.

Utility of structural subtyping. In a language with structural subtyping, such as Unity, not all interesting combinations of types have to be declared in advance (though in a library setting they might be, for consistency's sake). However, the key idea is that a type alias would simply be syntactic sugar for a set of methods, which could be given a different type alias in a different part of the system. Additionally, the subtyping relationships between all the interfaces would not need to be defined in advance. Finally, as a side note, the notational overhead in defining type aliases would be potentially far lower than that of defining a Java interface, which has a relatively high notational cost (due, in part to the nominal nature of interfaces).

In the FAQ for the Java collections API design [Sun Microsystems 2003], in explaining the rationale for the optional methods, examples are given for additional interfaces that would be useful. One example is that of logs, such as error logs and audit logs. As these are append-only sequences, they should support all of the List operations except for remove( ) and set( ) (replace value). For a Java implementation, this would require a new core interface, and a new iterator interface.

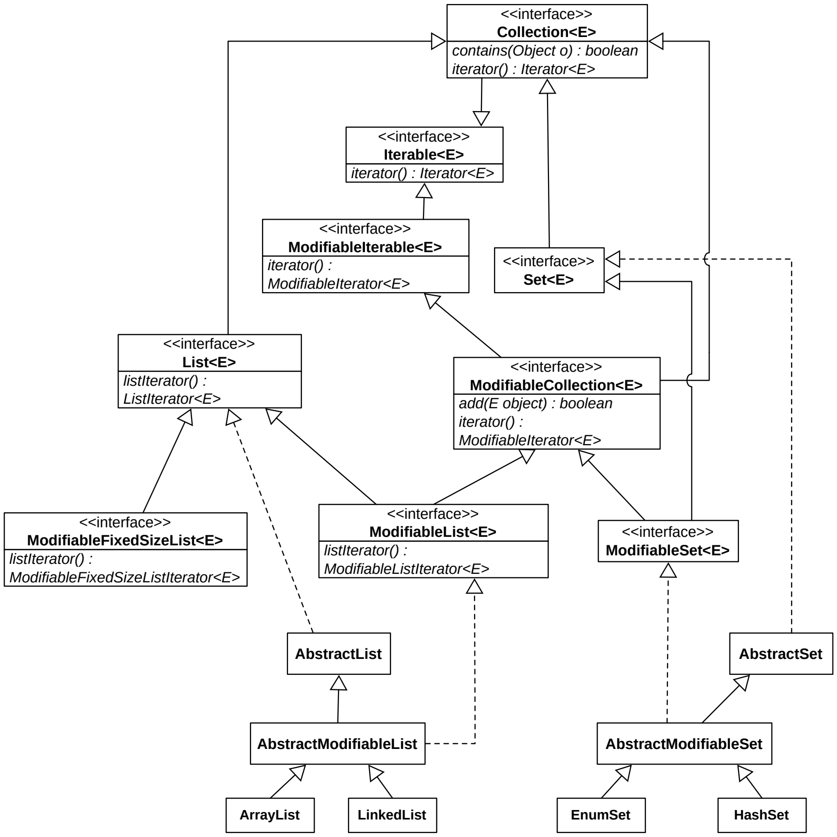

Figure 2.11: Refactored AbstractList and AbstractSet classes, along with new interfaces to remove optional methods. Only a subset of an interface's methods are listed. Type parameters are elided in classes.

Another example given in the FAQ is that of immutable collections—ones that cannot be modified by any client, not just through the current reference. This kind of type can be useful because it doesn't require synchronization. However, to support such invariants in the library, 4 additional core interfaces are need, plus additional iterator interfaces.

It is interesting to note that these two examples in the FAQ do not arise naturally from the design of the collections library—they are design considerations that might be useful. This highlights the mindset of the Java developer: since everything must be defined in advance, any potentially useful interface must be considered ahead of time and its advantages carefully weighed.

2.4 Methods in Unity

This section presents, at a high level, the properties of methods in the Unity formal system. I describe the semantics of method dispatch, rules for typechecking external methods, and the naming convention for internal and external methods.

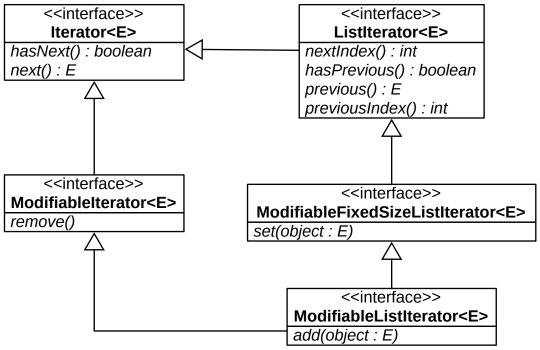

Figure 2.12: Refactored iterator interfaces. All methods, except for inherited methods, are shown.

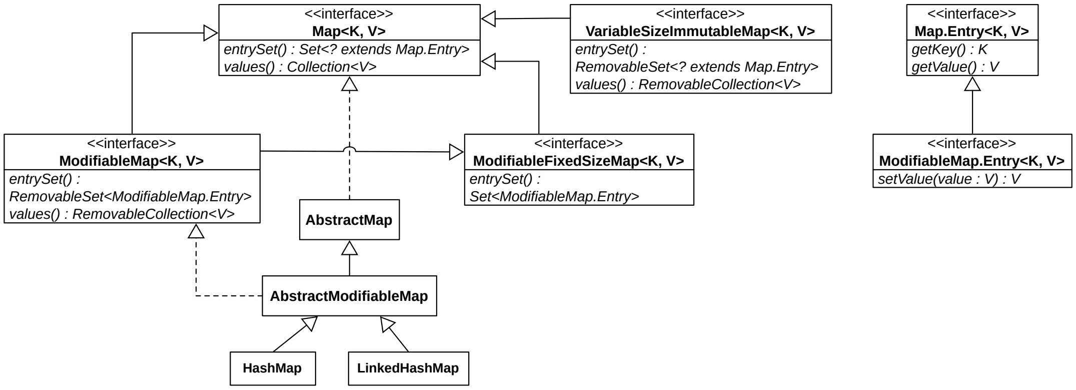

Figure 2.13: Refactored AbstractMap class, along with new interfaces to remove optional methods. Only a subset of an interface's methods are listed. Type parameters are elided in classes.

2.4.1 Dispatch Semantics

Methods may be defined with structural constraints, but these constraints are not used in dispatch—only brands are used. Thus, it is invalid to define two methods with the same name and that dispatch on the same brand, with differing structural constraints. Therefore, when overriding a method m, a subclass may not add structural constraints to m's receiver or its arguments. Similarly, it is not possible to providde two definitions for an external method $C.m$.

2.4.2 External Method Definitions

Recall that in Unity, external methods may be overridden by other methods, either internal or external. Typechecking an external method has two components: *exhaustiveness checking* (the provided cases provide full coverage of the dispatch hierarchy) and *ambiguity checking* (when executing a given method call, only one method is applicable).

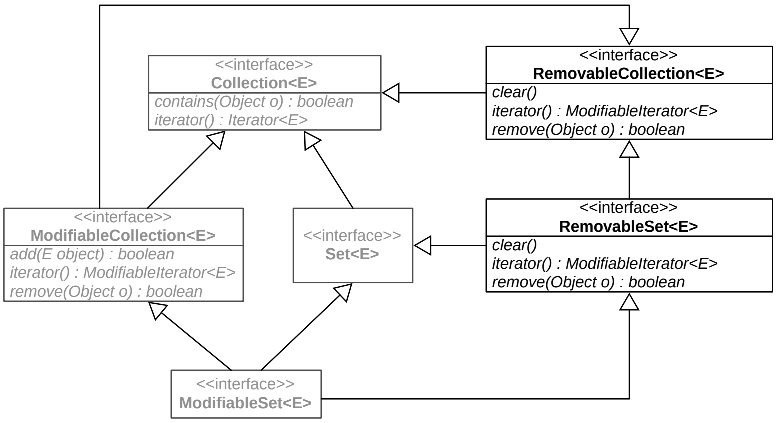

Figure 2.14: RemovableCollection and RemovableSet. Interfaces have been repeated from other figures to show subtyping relationships; these have been grayed out. Only a subset of an interface's methods are listed.

I have adapted the restrictions on external methods that were enforced by Millstein and Chambers' "System M" variant of the Dubious calculus [Millstein and Chambers 2002], and by later extensions such as MultiJava [Clifton et al. 2000] and EML [Millstein et al. 2002; 2004].

In Unity, exhaustiveness of external methods is ensured because there are no abstract methods. If such a feature were present, external method definitions would not be permitted to be abstract—just as in the aforementioned languages.

Additionally, to allow *modular* ambiguity checking, Unity methods must obey the following rules:

E1. All external method definitions of a method *m* must appear in the method block where the method family *m* is *introduced* (using the method declaration).

E2. An external method definition may *not* override an internal one (though an internal method *may* override an external one).

E3. When an external method family *m* is introduced, it must declare an *owner brand C*: this specifies that the method family is rooted at *C*. *C* must be a proper subtype of Object, the root of the inheritance hierarchy. An external method definition *m* for brand *D* is valid only if *D* is a sub-*brand* of *C*.

Here, a method *family* is defined as a method and all of its overrides. For internal methods, the overrides are spread across multiple classes, but for external methods, condition *E1* ensures that the external declarations in the family appear in the same syntactic block.

Condition *E1* is necessary because otherwise there could be two external method definitions *m* defined for the same brand *C*, leading to an ambiguity. This ambiguity would be impossible to detect in a modular manner, since when checking particular external method, the typechecker would have to do a non-modular search for other external methods with the same name. (Note that in the code examples, all external method definitions appeared together, though for brevity a method block was not used.)

Condition *E2* is required to avoid a situation where two external methods override the same internal method. Since neither external method is "better" than the other, this would result in a run-time method lookup ambiguity.

Finally, condition *E3* is necessary for ambiguity checking in the presence of multiple inheritance, and will be described further in Sect. 3.4.2. For this reason, the owner brand was omitted from the previous examples.

2.4.3 Simple and Qualified Method Names

Until now, I have glossed over the details of method naming, but as it turns out, to correctly implement condition *E2* (in a modular fashion), the formal system must have a way of distinguishing between external and internal method names.

Concretely, suppose we have an external method m defined on brand A. Now, we add brand B that extends A. Here, B is permitted to also define a method named m (with a different type, even) if the definition of $A.m$ is not in scope. Next suppose than an object o tagged with B is passed to a context where $A.m$ has been imported, and we call $o.m$. Now, the runtime semantics needs to be able to distinguish the external method from the internal method—otherwise either method is equally applicable. One straightforward way to achieve this is to internally use different names for each method.

Consequently, the calculus distinguishes between *qualified* names (denoted by the metavariable q) and *simple* names (denoted by n). The qualified name of a method family (i.e., a method and all of its overrides) is assumed to be globally unique. This can be easily implemented by generating a qualified name that includes the brand where the method family is first introduced. For example, an internal method m introduced in brand B (i.e., a new m declaration, rather than an override) could have simple name m and qualified name B_m. For external methods, the owner brand of an external method block can be used to generate the qualified name (e.g., $C\#m$).[12]

In order for structural subtyping to be useful, we must use simple names for the structural component of a type. That way, if two unrelated brands B and C each have methods with simple name n and type τ (and corresponding qualified names B_n and C_n), the simple name n should be used in so that objects of each brand are subtypes of $\mathsf{Object}(n : \tau)$. In contrast, if the qualified names were used, the only common supertype of B and C would be $\mathsf{Object}()$. Simple names are also needed for external methods, to support the pattern of adding a new external method $n : \tau$ so that objects of an existing brand D conform to type $D(n : \tau)$. An example of this, the newline method, was presented in Fig. 2.5.

Accordingly, in the calculus, each object contains a map from simple name to qualified name. This mapping can be either specified when the object is created, or can be added to an existing object via the with expression, described below.[13]

[12] This does not imply that owner brands are necessary for modular typechecking in the absence of multiple inheritance; an "owner" can always be automatically generated by taking the least upper bound of the brands on which the external method m is defined.

[13] Note that in the formal system, the simple and qualified names need not have any relation to one another, while in an actual implementation, the simple name would probably be a sub-string of the qualified name.

Only simple names appear in the structural component of a type; that is, types are of the form $B(\overline{n:\tau})$. If an object o has this type, the method call $o.n_i$ is a valid expression. Additionally, any internal or external method defined on B may be called on o, using qualified names (i.e., $o.q$).

Note that qualified method lookup can be easily implemented efficiently (e.g., using vtables), whereas simple name lookup must use the aforementioned map (and is more difficult to implement efficiently, due to depth and width subtyping). There is a benefit to this design: since simple names are useful primarily for the structural subtyping aspect of the system, structural subtyping becomes a "pay-as-you-go" feature. There is only a (potential) performance penalty if and when it is used.

Only the formal system includes both names; in the surface syntax, the programmer would only use the simple name and the qualified name would be automatically generated.[14] An elaboration phase (described below) would substitute simple names for qualified names where possible, and would otherwise generate "with" expressions (or, when possible, set up the simple-to-qualified mapping when objects are created). The using expression, which appears only in the surface syntax of the examples, would be used to determine which external methods should be used to generate the mapping.

A "with" expression must be generated whenever the typechecker must coerce an already-created object to a structural type. For instance, in Fig. 2.1, a BufferedReader object is coerced to Reader in order to call findString. The elaborator would report an error in the case where the same simple name can be mapped to two (or more) different qualified names.

Concretely, recall the highlighted lines of Fig. 2.1:

```
findString bufReader "foo"
findString charArrayReader "bar"
findString someOtherReader "baz"
```

Considering only the read method, this would be elaborated to:

```
findString (bufReader with read ↪ AbstractReader_read, . . . ) "foo"
findString (charArrayReader with read ↪ AbstractReader_read, . . . ) "bar"
findString (someOtherReader with read ↪ SomeOtherReader_read, . . . ) "baz".
```

Note that the with clause for charArrayReader is identical to that of bufReader; this is because read was introduced in AbstractReader.

Similarly, the highlighted line in Fig. 2.4,

```
readAndWrite stringStream
```

would be translated to

[14] Additionally, a scope qualifier construct would also be necessary to distinguish between two methods with the same simple name (which can occur either through multiple inheritance or importing of external method definitions). The explicit use of qualified names in the formal system sidesteps these issues.

```
readAndWrite
    (stringStream with
        read ↪ StringStream_read, . . . , // Reader methods
        append ↪ AbstractWriter_append, . . . , // Writer methods
        seek ↪ StringStream_seek
    ).
```

Finally, the highlighted line of Fig. 2.5,

```
new StringStream( ).writeLine "42"
```

translates to

```
new StringStream( newline ↪ AbstractWriter#newline ).AbstractWriter#writeLine "42".
```

Finally, note that in a system with state (my formal system is purely functional), the "with" construct would create a *wrapper* containing the new mapping along with a pointer to the old object. In such a system, there would be three notions of equality: wrapper equality ("==" in Java-like languages), reference equality, and value equality (equals( ) in Java). Wrapper equality would consider an object and its wrapped variant to be distinct, while reference equality would equate two wrappers that point to the same object but that perhaps have different simple-to-qualified name mappings. (This latter notion should perhaps be called "equality," as it does not preserve contextual equivalence. That is, if o_1 "unwrap-equals" o_2, this does not mean that $o_1.m$ and $o_2.m$ will call the same method m.[15] Regardless, I believe there are situations in which such a notion of "equality" could be useful.)

2.5 Formal System

The Unity grammar is presented in Fig. 2.15. The language is a lambda calculus with the addition of values tagged with brands. The metavariables B, C, D, and E range over brand names, and the overbar notation (e.g., $\overline{B}$) denotes a sequence of items (e.g., names, types, labels, etc.) that may be indexed by a variable. That is, $\overline{B}$ is equivalent to $B_i\ ^{i \in 1..x}$, where x is the length of the $\overline{B}$ sequence. The metavariables n and q ranges over simple and qualified method names, respectively; m ranges over both kinds of method names. M and N range over a sequence of (method : type) pairs (with simple names). There is a slight abuse of notation by using the set inclusion operator on lists (e.g., $m \in \overline{m}$), but the intended meaning should always be clear from context.

Portions of the formal system are highlighted —these are the aspects of the language pertaining to multiple inheritance. These features will be described in Sect. 3.7.

To define a brand, the brand top-level declaration is used. When a brand is defined, it is given a name, as well as the brand's *field type* (usually a record); this is the type of the fields of the brand. An object's field value is initialized when the object is created.

[15] That is to say, all unwrap-equal objects are created (quote-unquote) "equal," but some are more equal than others.

Programs	$p ::= \Sigma \rhd \mathit{decl}\ \mathsf{in}\ p \mid e$
Declarations	$\mathit{decl} ::= \mathit{brand\text{-}decl} \mid \mathit{method\text{-}decl}$
Brand declaration	$\mathit{brand\text{-}decl} ::= \mathsf{brand}\ B(\tau; \overline{\mathit{m\text{-}decl}})\ \mathsf{extends}\ C_1, \ldots, C_n\ \mathsf{requires}\ \overline{D}$
Method declaration	$\mathit{m\text{-}decl} ::= q\ B(\overline{n : \rho}) : \tau = e$
External method block	$\mathit{method\text{-}decl} ::= \mathsf{method}\ B\ .q(\overline{\mathit{m\text{-}decl}})$
Expression types	$\tau, \sigma ::= \mathsf{unit} \mid \tau \to \tau \mid \tau \wedge \tau \mid B(\overline{n : \rho}) \mid \{\overline{\ell : \tau}\} \mid X \mid \mu X.\tau$
Method types	$\rho ::= (\overline{n : \rho}) \Rightarrow \tau$
Expressions	$e ::= () \mid x \mid \lambda x{:}\tau.\, e \mid e\, e \mid \widehat{B}(e; \overline{n \hookrightarrow q}) \mid e\ \mathsf{with}\ \overline{n \hookrightarrow q} \mid (\overline{\ell = e}) \mid e.\ell \mid$
	$\mid e.m \mid e.B.\mathsf{super}.q \mid \mathsf{fold}_\tau\ e \mid \mathsf{unfold}_\tau\ e$
Values	$v ::= () \mid \widehat{B}(v; \overline{n \hookrightarrow q}) \mid (\overline{\ell = v}) \mid \lambda x{:}\tau.\, e \mid \mathsf{fold}_\tau\ v$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, X \leq Y$
	$\Sigma ::= \cdot \mid \Sigma, \mathit{decl\text{-}type}$
	$\mathit{decl\text{-}type} ::= \mathsf{brand}\ B(\tau; \overline{q : \rho})\ \mathsf{extends}\ \overline{C}\ \mathsf{requires}\ \overline{D}$
	$\mid \mathsf{method}\ B\ .q(\overline{C.q : \rho})$
	$\Delta ::= \cdot \mid \Delta, B(\overline{q = e})\ \mathsf{extends}\ \overline{C} \mid \mathsf{method}\ q(\overline{B.q = e})$

Definitions

$\text{fieldType}_\Sigma(B) \overset{\text{def}}{=} \tau \quad \text{where} \quad B(\tau; \ldots) \cdots \in \Sigma$

$B \overset{\text{def}}{=}$ type corresponding to tag $\widehat{B}$

ℓ, k range over record label names

B, C, D, E range over brand names

n ranges over “simple” (non-unique) method names

q ranges over qualified method names

m ranges over n, q

M, N range over $\overline{n : \rho}$

Q ranges over $\overline{q : \rho}$

r ranges over ℓ, k, m

t ranges over τ, ρ

Figure 2.15: Unity grammar. The portions relating to multiple inheritance are highlighted and will be discussed in Chapter 3.

Methods can be defined on a brand either externally or internally and the usual object-oriented method dispatch semantics apply.[16] Like brand declarations, external method declarations are top-level. Note that condition *E1* is enforced syntactically—all external method definitions must appear in the same block. To support multiple inheritance, method blocks also contain an owner brand; the need for this is discussed further in Chapter 3.

It is important to note that top-level declarations *decl* (brand or external method) are each declared with a context Σ. This is the context under which the brand or external method is to be typechecked; it must contain all declarations on which *decl* statically depends. This context is included for two reasons: first, in a real program, dependencies between modules are specified by the user (or inferred by the module system). Next, this eases the modularity proof (described in Sect. 3.7.3), which demonstrates that declarations can be separately typechecked under their declared context and no additional checks (analogous to link-time checking) is necessary.2

As previously described, methods take only one argument: the receiver (i.e., the this parameter). Additional parameters may be specified using lambda expressions. When a method q is defined for a brand B, the structural constraint M (a sequence of {method : type} pairs) is specified for the receiver: these are the methods that must exist as part of an object o's structural type in order for $o.q$ to be well-typed.

The calculus distinguishs between expression types (denoted by τ and σ) and method types (denoted by ρ). Method types use the arrow "$\Rightarrow$" while function types use arrow "$\rightarrow$".

The language includes a limited form of intersection types, which increase expressiveness and simplify some aspects of the formal system. Section 2.5.1 describes intersection types in more detail. Iso-recursive types are also included; these have the standard fold and unfold expressions with the usual static and dynamic semantics.

If B is a brand name, then $\widehat{B}$ is the tag value corresponding to B. Values also contain the simple-to-qualified mapping described in the previous section. To create objects, the expression form $\widehat{B}(e, \overline{n \hookrightarrow q})$ is used; this creates an object tagged with $\widehat{B}$ that has type $B(\overline{n : \tau})$, where each q_i has type τ_i.

To modify an object's mapping after it has been created, the with expression is used. This adds mappings $\overline{n \hookrightarrow q}$ to the object, replacing any existing mappings that have the same simple name and are associated with the object value.

Σ is the brand context; it stores the declared brand and method types. Δ is the corresponding run-time context. Δ retains method bodies but discards all type information (with the exception of brand tags). As in Featherweight Java [Igarashi et al. 1999], the system assumes the existence of a special brand Object that is not defined in Σ or Δ, but that may be extended by user-defined brands. Since every brand must have (at least) one super-brand, the brand subtype hierarchy is rooted at Object.[17]

Note that the context Σ and Δ are sometimes omitted from the rules where it is obvious from the manner in which they are used; in such cases, the required context is an unchanged "Σ" or

[16] Note that in the formal system, the syntax for internal method declarations mirrors that of external method declaration; in constrast, in the code examples, internal methods have syntax "$m : \rho = e$."

[17] Note that an alternative design is also possible: classes that extend no other class form a forest with no common ancestor. One advantage of this design is that then there would be no need for special-case rules for Object (e.g., premise (1) of TP-EXT-METHOD in Fig. 2.19).

$\boxed{B_1 \sqsubseteq_\Sigma B_2}$ **Sub-brand judgement**

$$\frac{}{B \sqsubseteq_\Sigma B}\ (\textsc{Sub-Brand-Refl}) \qquad \frac{B_1 \sqsubseteq_\Sigma B_2 \qquad B_2 \sqsubseteq_\Sigma B_3}{B_1 \sqsubseteq_\Sigma B_3}\ (\textsc{Sub-Brand-Trans}) \qquad \frac{B(\tau; Q)\ \mathsf{extends}\ C_1, \ldots, C_n \in \Sigma}{B \sqsubseteq_\Sigma C_i}\ (\textsc{Sub-Brand-Decl})$$

$\boxed{\Gamma \vdash_\Sigma \overline{r_a : t_a} \trianglelefteq \overline{r_b : t_b}}$ **Sub-row judgement**

$$\frac{r_i : t_i\ ^{i \in 1..n} \text{ is a permutation of } r_j : t_j\ ^{j \in 1..n}}{\Gamma \vdash r_i : t_i\ ^{i \in 1..n} \trianglelefteq r_j : t_j\ ^{j \in 1..n}}\ (\textsc{Sub-Row-Perm}) \qquad \frac{n > m}{\Gamma \vdash r_i : t_i\ ^{i \in 1..n} \trianglelefteq r_j : t_j\ ^{j \in 1..m}}\ (\textsc{Sub-Row-Width})$$

$$\frac{\Gamma \vdash t_i \leq t_i' \quad (i \in 1..n)}{\Gamma \vdash r_i : t_i\ ^{i \in 1..n} \trianglelefteq r_i : t_i'\ ^{i \in 1..n}}\ (\textsc{Sub-Row-Depth}) \qquad \frac{\Gamma \vdash \overline{r_a : t_a} \trianglelefteq \overline{r_b : t_b} \qquad \Gamma \vdash \overline{r_b : t_b} \trianglelefteq \overline{r_c : t_c}}{\Gamma \vdash \overline{r_a : t_a} \trianglelefteq \overline{r_c : t_c}}\ (\textsc{Sub-Row-Trans})$$

Figure 2.16: Sub-brand and sub-row judgements

"Δ". Judgements that are also functions (such as $mtype_\Sigma$, $mbody_\Delta$, etc.) are subscripted with the appropriate context, while other judgements use a subscript on the turnstile symbol (i.e., "$\vdash_\Sigma$").

Like many other object calculi, Unity is purely functional so as to simplify the formal system. State is orthogonal to the issues under consideration (with the exception of the with expression on imperative objects, which was described in Sect. 2.4.3). There does not appear to be any inherent reason why the language design could not be adapted to a language with imperative features.

The static and dynamic semantics of Unity (excluding multiple inheritance features) are described below; a discussion of the proof of type safety appears in Sect. 3.7.4 (with full proof in Appendix A). Additionally, a proof of the modularity of the type system is presented in Sect. 3.7.3.

2.5.1 Static Semantics

In this section, the subtyping and typing judgements shown in Figs. 2.16, 2.17, 2.18 and 2.21 are described. Auxiliary judgements are in Figs. 2.20 and 2.22.

Subtyping

Unity contains three distinct sub-"type" relations: sub-brand ($\sqsubseteq_\Sigma$), sub-row ($\trianglelefteq$) and subtype ($\leq$). Sub-branding (Fig. 2.16) is not on types but rather brands, which are a component of an object type. This relation is simply the reflexive, transitive closure of the declared extends relation.

The sub-row relation corresponds to Definition 2.2: it is the structural part of the system and includes depth and width subtyping. This relation is used for both record subtyping and method list subtyping: rules Sub-Record, Sub-Obj and Sub-Method of Fig. 2.17.

$\boxed{\Gamma \vdash_\Sigma \tau_1 \leq \tau_2}$ **Subtype judgement (expression types)**

$$\text{(Sub-Refl)}\ \frac{}{\Gamma \vdash \tau \leq \tau} \qquad \text{(Sub-Trans)}\ \frac{\Gamma \vdash \tau_1 \leq \tau_2 \qquad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3} \qquad \text{(Sub-Func)}\ \frac{\Gamma \vdash \sigma_1 \leq \tau_1 \qquad \Gamma \vdash \tau_2 \leq \sigma_2}{\Gamma \vdash \tau_1 \to \tau_2 \ \leq\ \sigma_1 \to \sigma_2}$$

$$\text{(Sub-}\wedge R\text{)}\ \frac{\Gamma \vdash \tau \leq \sigma_1 \qquad \Gamma \vdash \tau \leq \sigma_2}{\Gamma \vdash \tau \leq \sigma_1 \wedge \sigma_2} \qquad \text{(Sub-}\wedge L_1\text{)}\ \frac{}{\Gamma \vdash \tau_1 \wedge \tau_2 \leq \tau_1} \qquad \text{(Sub-}\wedge L_2\text{)}\ \frac{}{\Gamma \vdash \tau_1 \wedge \tau_2 \leq \tau_2}$$

$$\text{(Sub-Record)}\ \frac{\Gamma \vdash \overline{\ell : \tau} \trianglelefteq \overline{k : \sigma}}{\Gamma \vdash \{\overline{\ell : \tau}\} \leq \{\overline{k : \sigma}\}} \qquad \text{(Sub-Type-Var)}\ \frac{X \leq Y \in \Gamma}{\Gamma \vdash X \leq Y} \qquad \text{(Sub-Mu)}\ \frac{\Gamma, X \leq Y \vdash \tau_1 \leq \tau_2}{\Gamma \vdash \mu X.\tau_1 \leq \mu Y.\tau_2}$$

$$\text{(Sub-Obj)}\ \frac{B_1 \sqsubseteq_\Sigma B_2 \qquad \Gamma \vdash M_1 \trianglelefteq M_2}{\Gamma \vdash B_1(M_1) \leq B_2(M_2)} \qquad \text{(Sub-Requires)}\ \frac{\begin{array}{c} B(\tau; Q)\ \mathsf{extends}\ \overline{B}\ \mathsf{requires}\ C_1, \ldots, C_n \in \Sigma \\ \Gamma \vdash M_1 \trianglelefteq M_2 \end{array}}{\Gamma \vdash B(M_1) \leq C_i(M_2)}$$

$\boxed{\Gamma \vdash_\Sigma \rho_1 \leq \rho_2}$ **Subtype judgement (method types)**

$$\text{(Sub-Method)}\ \frac{\Gamma \vdash M_2 \trianglelefteq M_1 \qquad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash M_1 \Rightarrow \tau_1 \ \leq\ M_2 \Rightarrow \tau_2}$$

Figure 2.17: Subtype judgements. The context Σ is omitted from the turnstile symbol in the inference rules, as it does not change in the course of the derivation (i.e., the rules use whichever context was passed to the judgement)

The rule Sub-Record is straightforward; it simply uses the sub-row judgement directly. The rule Sub-Obj corresponds to Definition 2.1 and uses both the sub-brand and sub-row judgements. This rule specifies that an object type $B_1(M_1)$ is a subtype of $B_2(M_2)$ when B_1 is a sub-brand of B_2 ($B_1 \sqsubseteq_\Sigma B_2$) and M_1 is a sub-row of M_2 ($M_1 \trianglelefteq M_2$).

For subtyping on method types, the rule Sub-Method is a straightforward generalization of function subtyping: it is contravariant in the argument position (the structural constraints on the receiver) and covariant in the result.

The language also includes a limited form of intersection types, à la Davies and Pfenning (i.e., no distributivity rule); the rules for subtyping intersection types are borrowed from their work [Davies and Pfenning 2000].

Aside from Sub-Requires, which will be described in Sect. 3.7, the remaining subtyping rules are standard.

Typechecking Programs

Full typing rules for typechecking programs and expressions appear in Figs. 2.18 and 2.21, respectively. Auxiliary judgements for typechecking programs are in Figures 2.19 and 2.20; other auxiliary judgements appear in 2.22

There are two typing rules for checking programs; these appear in Fig. 2.18. When typechecking a program, the type information corresponding to each contained declaration is added to the main context Σ, which is used to typecheck subsequent declarations.

For checking a brand or external method declaration, the rule Tp-Decl-Ok is used. This first uses the auxiliary judgement *decl* : *decl-type* to extract the declaration's declared type. Next, premise (2) checks that the declaration name (either brand or external method) does not already exist in the program context Σ. Premise (3) typechecks the types in the declaration *under its declared context* Σ_0, using the *decl-type* **ok** judgement described below. The method bodies in the declaration (either internal or external methods) are next typechecked under Σ_0, *decl-type* (to allow recursive methods) using the **body-ok**judgement (also described below).

Here, it is of key importance that declarations and method bodies are typechecked under Σ_0 rather than Σ, as otherwise it would be very difficult to prove that typechecking is modular. (The details of the modularity proof are described in Sect. 3.7.3.)

The next check, premise (5), ensures that the main context contains at least all of the declarations assumed by the declaration *decl*. Here, we use ordinary set inclusion.[18] Finally, the rest of the program is typechecked under Σ extended with the new declaration type.

Brand declarations. The rule Tp-Brand-Decl (Fig. 2.19) ensures that a brand declaration B is well-formed. First, we check that the inheritance structure of B is correct, using the **inherit-ok**auxiliary judgement (Fig. 2.20), which is described in Sect. 3.7.

Premise (2) of Tp-Brand-Decl creates a new context extended with the current declaration, and premise (3) checks that newly defined brand contains at least the fields of the supertype (possibly with refined types), under this new context.[19]

Finally, premise (4) checks that there are no duplicate internal methods in B and premise (5) checks that each method is a valid override of the same method in superclasses, should such a method exists. This last premise uses the **override-ok**judgement in Fig. 2.20. The rule for method overriding is a straightforward generalization of that of Featherweight Java; the overriding method type ρ must be a subtype of each of ρ_i, the types of the super-brand methods being overridden (if such methods exist). This rule uses the *mtype* auxiliary judgement, which

[18] Note that it would *not* be sound to take into account some form of subtyping on the declarations contained in Σ to perform this check. This is because the program may make either covariant or contravariant uses of the declarations within Σ. For instance, when making method calls on an object of a particular class C, it would be sound to substitute some C' where $C \sqsubseteq_\Sigma C'$ for the purpose of typechecking; i.e., this is a covariant use of C. In contrast, when extending a class, this corresponds to a contravariant usage. Since the type of usage of each declaration is not specified when declaring Σ_0, we must assume either covariant or contravariant uses may occur; therefore the subset relation must be invariant.

[19] As a consequence, if the field type is a record, sub-brands must list all the labels of the parent (and may therefore access them). Aside from simplifying the calculus, this sidesteps issues of variable shadowing while allowing subtypes to refine the type of a particular label.

$\boxed{\Sigma_1 \supseteq \Sigma_2}$ $\quad \supseteq \overset{\text{def}}{=}$ standard set inclusion

$\boxed{\Sigma \vdash p\ \mathbf{ok}}$

(Tp-Decl-Ok)

$$\frac{\begin{array}{c} \textcircled{1}\ decl : decl\text{-}type \qquad \textcircled{2}\ decl\text{-}type_{name} \notin \Sigma \\ \textcircled{3}\ \Sigma_0 \vdash decl\text{-}type\ \mathbf{ok} \qquad \textcircled{4}\ \Sigma_0, decl\text{-}type \vdash decl\ \mathbf{body\text{-}ok} \\ \textcircled{5}\ \Sigma \supseteq \Sigma_0 \qquad \textcircled{6}\ \Sigma, decl\text{-}type \vdash p\ \mathbf{ok} \end{array}}{\Sigma \vdash \Sigma_0 \rhd decl\ \text{in}\ p\ \mathbf{ok}}$$

(Tp-Expr-Ok)

$$\frac{\cdot \vdash_{\Sigma} e : \tau}{\Sigma \vdash e\ \mathbf{ok}}$$

$\boxed{decl : decl\text{-}type}$

$$\frac{}{\begin{array}{c} \text{brand } B(\sigma; q_i\ B(M_i) : \tau_i = e_i\ ^{i\in 1..n}) \text{ extends } \overline{C}\ \text{ requires } \overline{D} : \\ \text{brand } B(\sigma; q_i : M_i \Rightarrow \tau_i\ ^{i\in 1..n}) \text{ extends } \overline{C}\ \text{ requires } \overline{D} \end{array}}$$

$$\frac{}{\text{method } B.q(q\ C_i(M_i) : \tau_i = e_i\ ^{i\in 1..n})\ :\ \text{method } B.q(C_i.q : M_i \Rightarrow \tau_i\ ^{i\in 1..n})}$$

$\boxed{decl\text{-}type_{name}}$

$$\frac{}{\text{brand } B(\ldots)\ \ldots\ _{name} = B} \qquad \frac{}{\text{method } B.q(\ldots)_{name} = q}$$

Figure 2.18: Typechecking programs

looks up the type of the method (internal or external) defined or inherited in the specified brand. (*mtype* appears in Fig. 2.22 and is described in a subsection below.)

For typechecking internal method bodies, the rule Brand-Decl-Body is used, which checks that each method body has the correct type, when the special variables this and fields are bound to the appropriate types. Note that here the method's structural constraints M_i are added to the brand's nominal type B by specifying the type $B(M_i)$ for the this variable. Any brand methods (internal or external) can still be accessed by using a qualified name, in the case that a method name in the structural constraint conflicts with an existing method name.

External method declarations. The rule Tp-Ext-Method (Fig. 2.19) checks external method definitions. Premise (2) ensures that there are no duplicate external method definitions (i.e., more than one method defined for the same brand C). Premise (4) enforces condition *E2*: an external method may not override an internal method. The $B.q$ **internal** auxiliary judgement is used here (Fig. 2.20); this simply searches for a method q that is either defined internally in B or is inherited. Finally, as in the rule for brand declarations, premise (6) ensures that each external method definition is a valid override of other (external) methods, using the **override-**

$$\boxed{\Sigma \vdash \textit{decl-type}\ \mathbf{ok}}$$

(Tp-Brand-Decl)

$$\frac{\begin{array}{c}\textcircled{1}\ \vdash_\Sigma B \text{ extends } \overline{C} \text{ requires } \overline{D}\ \mathbf{inherit\text{-}ok}\\ \textcircled{2}\ \Sigma' = \Sigma, \text{brand } B(\tau;\overline{q:\rho}) \text{ extends } \overline{C} \text{ requires } \overline{D} \qquad \textcircled{3}\ \vdash_{\Sigma'} \tau \le \text{fieldType}_\Sigma(\overline{C})\\ \textcircled{4}\ \overline{q} \text{ distinct} \qquad \textcircled{5}\ \vdash_{\Sigma'} B.\overline{q:\rho}\ \mathbf{override\text{-}ok}\end{array}}{\Sigma \vdash \text{brand } B(\tau;\overline{q:\rho}) \text{ extends } \overline{C} \text{ requires } \overline{D}\ \mathbf{ok}}$$

(Tp-Ext-Method)

$$\frac{\begin{array}{c}\textcircled{1}\ B \neq \text{Object} \qquad \textcircled{2}\ \overline{C} \text{ distinct} \qquad \textcircled{3}\ C_i \sqsubseteq_\Sigma B \ \ (\forall i \in 1..n)\\ \textcircled{4}\ \nvdash_\Sigma \overline{C}.q\ \mathbf{internal} \qquad \textcircled{5}\ \Sigma' = \Sigma, \text{method } B.q(\overline{C}.q:\overline{\rho}) \qquad \textcircled{6}\ \vdash_{\Sigma'} \overline{C}.q:\overline{\rho}\ \mathbf{override\text{-}ok}\end{array}}{\Sigma \vdash \text{method } B.q(\overline{C}.q:\overline{\rho})\ \mathbf{ok}}$$

$$\boxed{\Sigma \vdash \textit{decl}\ \mathbf{body\text{-}ok}}$$

(Brand-Decl-Body)

$$\frac{\text{fieldType}_\Sigma(\overline{D}) = \overline{\sigma}' \qquad \text{this}: B(M_i), \text{fields}: \sigma \wedge \overline{\sigma}' \vdash_\Sigma e_i : \tau_i}{\text{brand } B(\sigma; q_i\ B(M_i) : \tau_i = e_i\ {}^{i \in 1..n}) \text{ extends } \overline{C} \text{ requires } \overline{D}\ \mathbf{body\text{-}ok}}$$

(Ext-Method-Body)

$$\frac{\text{this}: C_i(M_i) \vdash_\Sigma e_i : \tau_i \ \ (\forall i \in 1..n)}{\Sigma \vdash \text{method } B.q(q\ C_i(M_i) : \tau_i = e_i\ {}^{i \in 1..n})\ \mathbf{body\text{-}ok}}$$

Figure 2.19: Typechecking brand and method declarations

okjudgement.

For typechecking external method bodies, rule Ext-Method-Body applies, which is similar to the rule for typechecking internal methods. The sole difference is that the fields variable is not bound in Γ and therefore cannot be accessed by the method body, which effectively enforces a form of information hiding.

Expressions. Typechecking expressions (Tp-Expr-Ok) simply defers to the judgement for typechecking expressions (Fig. 2.21). Most of the rules are standard; the exceptions are Tp-New-Obj, Tp-With, Tp-Invoke-Struct, Tp-Invoke-Nom, and Tp-Invoke-Super. The first four are described here; the last regards multiple inheritance and is described in Sect. 3.7.

The rule Tp-New-Obj checks the correctness of the object creation expression. First, we check that the provided expression conforms to the field type of the brand. Next, we check that the simple names provided in the mapping are distinct, then look up the method type ρ_i of each qualified name (using the *mtype* judgement, described below). The resulting type has a structural component that maps each n_i to the corresponding ρ_i.

$\boxed{\vdash_\Sigma B.q : \rho\ \textbf{override-ok}}$

$$\frac{B \text{ extends } C_1, \ldots, C_n \in \Sigma \qquad mtype_\Sigma(q, C_i) = \rho'_i \text{ implies } \rho \leq \rho'_i \quad (\forall i \in 1..n)}{\vdash_\Sigma B.q : \rho\ \textbf{override-ok}}$$

$\boxed{\vdash_\Sigma B\ \textbf{inherit-ok}}$

(Tp-Inherit)

$$\frac{\begin{array}{c} ① \ \overline{C} \in \Sigma \qquad ② \ \overline{D} \in \Sigma \qquad ③ \ D_i \notin \overline{C} \ \ (\forall i \in 1..m) \\ ④ \ \forall i, j.\, i \neq j.\, \nexists D'.\, C_i \sqsubseteq_\Sigma D' \text{ and } C_j \sqsubseteq_\Sigma D' \quad (D' \neq \text{Object}) \\ ⑤ \ C_i \text{ requires } E \in \Sigma \ \text{ implies } \ \exists k.\ C_k \sqsubseteq_\Sigma E \text{ or } D_k \sqsubseteq_\Sigma E \ \ (\forall i \in 1..n) \\ ⑥ \ D_i \text{ requires } E' \in \Sigma \ \text{ implies } \ \exists k.\ C_k \sqsubseteq_\Sigma E' \text{ or } D_k \sqsubseteq_\Sigma E' \ \ (\forall i \in 1..m) \\ ⑦ \ \forall i, j.\, \forall q.\, mtype_\Sigma(q, C_i) = \rho \text{ and } mtype_\Sigma(q, C_j) = \rho' \text{ implies } i = j \end{array}}{\vdash_\Sigma B \text{ extends } C_1, \ldots, C_n \text{ requires } D_1, \ldots, D_m \in \Sigma\ \ \textbf{inherit-ok}}$$

$\boxed{\vdash_\Sigma B.q\ \textbf{internal}}$

(Internal-Base)

$$\frac{\text{brand } B(\tau; \ldots, q : \rho, \ldots) \in \Sigma}{\vdash_\Sigma B.q\ \textbf{internal}}$$

(Internal-Inh)

$$\frac{\begin{array}{c} \text{brand } B(\tau; Q) \text{ extends } \overline{C} \in \Sigma \qquad q \notin Q \\ \exists k.\, C_k.q\ \textbf{internal} \end{array}}{\vdash_\Sigma B.q\ \textbf{internal}}$$

Figure 2.20: Auxiliary judgements for typechecking programs

The *mtype* auxiliary judgement (defined in Fig. 2.22) looks up the type of a qualified name defined for a particular brand B. The judgement first checks for internal methods; if one does not exist, the second rule searches for an external definition. If neither of these cases applies, we perform a lookup using some super-brand of B.

The next rule, Tp-With, performs an operation similar to that of Tp-New-Obj. The names in the specified map must not be contained in the expression's structural type ($\overline{n} \notin M$) and must be distinct. Then *mtype* is used to lookup the types of the corresponding qualified names. There is one difference between Tp-With and Tp-New-Obj, involving required brands (premises 2 and 5). This difference is described further in Sect. 3.7.

It should be noted that the subsumption rule can always be used to "forget" methods in an object's structural type, so it is possible add a simple name mapping $n \hookrightarrow q$ that already existed in the object's map. That is, the situation depicted in Fig. 2.23 can occur. Since the subsumption rule allows the typechecker to "forget" that e_1 has method n, the "e_1 with …" expression is still well-typed. At runtime, the new mapping replaces the old one.

Typechecking method invocations (rules Tp-Invoke-Struct and Tp-Invoke-Nom) is fairly straightforward, as the body of the method was already checked when the method was declared. When invoking a method with a simple name (structural method invocation), we ensure that the method n exists in the structural part of the expression's type (M) and has type $N \Rightarrow \tau$. Finally, M

$\boxed{\Gamma \vdash_\Sigma e : \tau}$

(Tp-Var)
$$\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

(Tp-Unit)
$$\dfrac{}{\Gamma \vdash () : \mathsf{unit}}$$

(Tp-Fun)
$$\dfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1 .\, e : \tau_1 \to \tau_2}$$

(Tp-App)
$$\dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

(Tp-Subs)
$$\dfrac{\Gamma \vdash e : \sigma \qquad \vdash_\Sigma \sigma \le \tau}{\Gamma \vdash e : \tau}$$

(Tp-New-Record)
$$\dfrac{\Gamma \vdash \overline{e} : \overline{\tau}}{\Gamma \vdash (\overline{\ell = e}) : \{\overline{\ell : \tau}\}}$$

(Tp-Proj)
$$\dfrac{\Gamma \vdash e : \{\ell_i : \tau_i{}^{i \in 1..n}\}}{\Gamma \vdash e.\ell_k : \tau_k}$$

(Tp-New-Obj)
$$\dfrac{B \text{ requires } \bullet \in \Sigma \qquad \mathrm{fieldType}_\Sigma(B) = \tau \qquad \Gamma \vdash e : \tau \qquad \overline{n} \text{ distinct} \qquad \mathit{mtype}_\Sigma(\overline{q}, B) : \overline{\rho}}{\Gamma \vdash \widehat{B}(e; \overline{n \hookrightarrow q}) : B(\overline{n : \rho})}$$

(Tp-With)
$$\dfrac{\Gamma \vdash e : B(M) \qquad B \text{ requires } \overline{D} \in \Sigma \qquad \overline{n} \notin M \qquad \overline{n} \text{ distinct} \qquad \exists C \in \{B, \overline{D}\}.\, \mathit{mtype}_\Sigma(q_i, C) : \rho_i \quad (\forall i \in 1..n)}{\Gamma \vdash e \text{ with } n_i \hookrightarrow q_i{}^{i \in 1..n} : B(M, n_i : \rho_i{}^{i \in 1..n})}$$

(Tp-Invoke-Struct)
$$\dfrac{\Gamma \vdash e : B(M) \qquad n : N \Rightarrow \tau \in M \qquad M \trianglelefteq N}{\Gamma \vdash e.n : \tau}$$

(Tp-Invoke-Nom)
$$\dfrac{\Gamma \vdash e : B(M) \qquad \mathit{mtype}_\Sigma(q, B) : N \Rightarrow \tau \qquad M \trianglelefteq N}{\Gamma \vdash e.q : \tau}$$

(Tp-Invoke-Super)
$$\dfrac{\Gamma \vdash e : B(M) \qquad B \text{ requires } C \in \Sigma \qquad \mathit{mtype}_\Sigma(q, C) : N \Rightarrow \tau \qquad M \trianglelefteq N}{\Gamma \vdash e.C.\mathsf{super}.q : \tau}$$

(Tp-Fold)
$$\dfrac{\Gamma \vdash e : [\mu X.\tau / X]\tau}{\Gamma \vdash \mathsf{fold}_{\mu X.\tau}\, e : \mu X.\tau}$$

(Tp-Unfold)
$$\dfrac{\Gamma \vdash e : \mu X.\tau}{\Gamma \vdash \mathsf{unfold}_{\mu X.\tau}\, e : [\mu X.\tau / X]\tau}$$

Figure 2.21: Typechecking expressions. The context Σ is omitted from the turnstile symbol in the inference rules, as it does not change in the course of the derivation (i.e., the rules use whichever context was passed to the judgement)

must be a structural subtype of *n*'s structural constraint ($M \trianglelefteq N$). The entire expression therefore has type τ.

For invoking a method with a qualified name (Tp-Invoke-Nom), the difference is that the method type is not looked up within *M*, but rather on the brand *B* using *mtype*. As before, the structural constraints must be satisfied ($M \trianglelefteq N$).

2.5.2 Dynamic Semantics

The evaluation judgements for programs and expressions appear in Figures 2.24 and 2.25, respectively. Auxiliary judgements for method lookup are in Fig. 2.26.

In evaluating programs (Fig. 2.24), the interesting rules are E-Brand-Decl and E-Ext-Decl, which evaluate brand definitions and external method definitions, respectively. To evaluate a brand definition, the method definitions are reduced to just the method name and body; the

$$\boxed{mtype_\Sigma(q,B)=\rho}$$

$$\text{(MType-Base)}\quad \frac{\textsf{brand } B(\tau;\ldots,q:\rho,\ldots)\in\Sigma}{mtype_\Sigma(q,B)=\rho} \qquad \text{(MType-Ext)}\quad \frac{\begin{array}{c}\textsf{brand } B(\tau;Q)\in\Sigma\\ extDef_\Sigma(q,B)=\rho\end{array}}{mtype_\Sigma(q,B)=\rho} \qquad \text{(MType-Inh)}\quad \frac{\begin{array}{c}\textsf{brand } B(\tau;Q) \textsf{ extends } \overline{C}\in\Sigma\\ q\notin Q \qquad \nexists\rho'.\, extDef_\Sigma(q,B)=\rho'\\ \exists k.\ mtype_\Sigma(q,C_k)=\rho\end{array}}{mtype_\Sigma(q,B)=\rho}$$

$$\boxed{extDef_\Sigma(q,B)=q:\rho}$$

$$\frac{\textsf{method } C.q(\ldots,B.q:\rho,\ldots)\in\Sigma}{extDef_\Sigma(q,B)=\rho}$$

Figure 2.22: Auxiliary functions for typechecking expressions

Program	Types	Values
	brand $B(\cdot;\ q_1:()\Rightarrow\tau_1, q_2:()\Rightarrow\tau_2)\in\Sigma$	
$e_1=\widehat{B}(\cdot, n\hookrightarrow q_1)$	$e_1:B(n:()\Rightarrow\tau_1)$	$e_1=\widehat{B}(\cdot, n\hookrightarrow q_1)$
$x_1=e_1.q_1$ $y_1=e_1.n$	$x_1:\tau_1$ $y_1:\tau_1$	$x_1\longmapsto v_1$ $y_1\longmapsto v_1$
$e_2=e_1$ with $n\hookrightarrow q_2$	$e_1:B(n:()\Rightarrow\tau_1)$ $B(n:()\Rightarrow\tau_1)\leq B()$ $e_1:B(n:()\Rightarrow\tau_1)$ $e_2:B(n:()\Rightarrow\tau_2)$	$e_2=e_1$ with $n\hookrightarrow q_2\longmapsto\widehat{B}(\cdot, n\hookrightarrow q_2)$
$x_2=e_2.q_2$ $y_2=e_2.n$	$x_2:\tau_2$ $y_2:\tau_2$	$x_2\longmapsto v_2$ $y_2\longmapsto v_2$

Figure 2.23: Example: typechecking new object and "with" expressions

rest of the program is evaluated with the extended context. Similarly, E-Ext-Decl updates the context with new external method definitions, then evaluates the rest of the program with the new context. In both of these rules, method declarations are evaluated by discarding all type information, leaving just the method name and body.

For evaluating expressions, the rules are are standard (or are congruence rules) except for E-Invoke-Val, E-Super-Invk-Val (which will be described in Sect. 3.7), and E-With-Val.

E-Invoke-Val uses the auxiliary judgement $mbody_\Delta(m,B,\overline{n\hookrightarrow q})$. This latter judgement, which I describe below, finds the appropriate method body e for a method m. The object is then substituted for this and its fields are substituted for fields within e.

The *mbody* judgement (Fig. 2.26) is itself straightforward; if we are performing qualified method lookup, the *lookup* judgement is called directly, otherwise the appropriate qualified name is found within the object's map.

Evaluating programs

$\boxed{p \mid \Delta \longmapsto p' \mid \Delta'}$

(E-Brand-Decl)

$$\frac{\overline{m\text{-}decl} \longmapsto \overline{B.q = e}}{\Sigma \rhd \text{ brand } B(\tau; \overline{m\text{-}decl}) \text{ extends } \overline{C} \text{ requires } \overline{D} \text{ in } p \mid \Delta \longmapsto p \mid \Delta, B(\overline{q = e}) \text{ extends } \overline{C}}$$

(E-Ext-Decl)

$$\frac{\overline{m\text{-}decl} \longmapsto \overline{C.q = e}}{\Sigma \rhd \text{method } B.q(\overline{m\text{-}decl}) \text{ in } p \mid \Delta \longmapsto p \mid \Delta, \text{method } q(\overline{C.q = e})}$$

(E-Expr)

$$\frac{e \longmapsto_{\Delta} e'}{\Sigma \rhd e \mid \Delta \longmapsto e' \mid \Delta}$$

Auxiliary judgement

$\boxed{m\text{-}decl \longmapsto q = e}$

$$\frac{}{q\ B(M) : \tau = e \ \longmapsto\ B.q = e}$$

Figure 2.24: Evaluation and auxiliary judgement for programs

The *lookup* judgement, in turn, is similar to *mbody* in Featherweight Java, with the exception that the rule Lookup-Ext is added (for external method lookup). The judgement first looks for methods defined within the brand itself (Lookup-Base). If such a method does not exist, an exactly matching external method (Lookup-Ext) is searched. If neither of these cases apply, we perform lookup on the unique super-brand of B for which *lookup* is defined.

Returning to expression evaluation, the rule E-With-Val evaluates a "with" expression by creating a new object with a combination of the new and old mappings. Here, the *merge* function is used (Fig. 2.26), which retains only only those old mappings $\overline{n_a \hookrightarrow q_a}$ that do not exist in the new map $\overline{n_b \hookrightarrow q_b}$. We saw an example of this in Fig. 2.23; the expression e_2 evaluated to an object containing only the new mapping for n (i.e., dropping the mapping $n \hookrightarrow q$).

2.5.3 Adding Polymorphism

The Unity formal system (as presented in this chapter) does not contain parametric polymorphism. This omission is intentional and is to simplify the formal system. Upon adding polymorphism to an earlier version of the calculus [Malayeri and Aldrich 2008a;b], I discovered that it was orthogonal to Unity's novel features.

This calculus, Unity$_\alpha$ adds type variables to brand declarations and includes polymorphic functions (defined the standard way, with the Λ notation). The calculus adds an additional judgement that checks that types are well-formed (e.g., a type instantiation must have the same arity as that of its corresponding type constructor). Unity$_\alpha$ was a straightforward extension of Unity, as there was not an interesting interaction between parametric polymorphism and either structural subtyping or external dispatch.

$\boxed{e \longmapsto_\Delta e'}$

Computation

(E-App-Abs)
$$\frac{}{(\lambda x{:}\tau.\, e)\ v \longmapsto_\Delta \{v/x\}\, e}$$

(E-Proj-Val)
$$\frac{}{(\ell_i = v_i\ ^{i\in 1..n}).\ell_k \longmapsto_\Delta v_k}$$

(E-Invoke-Val)
$$\frac{\exists \text{ unique } e.\ mbody_\Delta(m, B, \overline{n \hookrightarrow q}) = e}{\widehat{B}(v; \overline{n \hookrightarrow q}).m \longmapsto_\Delta \big\{\widehat{B}(v; \overline{n \hookrightarrow q})/\mathsf{this}, v/\mathsf{fields}\big\}\ e}$$

(E-Super-Invk-Val)
$$\frac{\exists \text{ unique } D.\ super_\Delta(B \text{ as } C) = D \qquad \exists \text{ unique } e.\ lookup_\Delta(q, D) = e}{\widehat{B}(v; \overline{n \hookrightarrow q}).C.\mathsf{super}.q \longmapsto_\Delta \big\{\widehat{B}(v; \overline{n \hookrightarrow q})/\mathsf{this}, v/\mathsf{fields}\big\}\ e}$$

(E-With-Val)
$$\frac{}{\widehat{B}\big(v; \overline{n_a \hookrightarrow q_a}\big) \text{ with } \overline{n_b \hookrightarrow q_b} \longmapsto_\Delta \widehat{B}\big(v;\ merge(\overline{n_b \hookrightarrow q_b}, \overline{n_a \hookrightarrow q_a})\big)}$$

(E-Unfold-Fold)
$$\frac{}{\mathsf{unfold}_\tau\ (\mathsf{fold}_\sigma\ v) \longmapsto_\Delta v}$$

Congruence

(E-App1)
$$\frac{e_1 \longmapsto_\Delta e_1'}{e_1\, e_2 \longmapsto_\Delta e_1'\, e_2}$$

(E-App2)
$$\frac{e_2 \longmapsto_\Delta e_2'}{v_1\, e_2 \longmapsto_\Delta v_1\, e_2'}$$

(E-Record)
$$\frac{e_k \longmapsto_\Delta e_k'}{(\ell_1 = v_1, \ldots, \ell_{k-1} = v_{k-1}, \ell_k = e_k, \ldots) \longmapsto_\Delta (\ldots, \ell_k = e_k', \ldots)}$$

(E-Proj)
$$\frac{e \longmapsto_\Delta e'}{e.\ell \longmapsto_\Delta e'.\ell}$$

(E-Obj)
$$\frac{e \longmapsto_\Delta e'}{\widehat{B}(e; \overline{n \hookrightarrow q}) \longmapsto_\Delta \widehat{B}(e'; \overline{n \hookrightarrow q})}$$

(E-Invoke)
$$\frac{e \longmapsto_\Delta e'}{e.m \longmapsto_\Delta e'.m}$$

(E-Super-Invk)
$$\frac{e \longmapsto_\Delta e'}{e.B.\mathsf{super}.q \longmapsto_\Delta e'.B.\mathsf{super}.q}$$

(E-With)
$$\frac{e \longmapsto_\Delta e'}{e \text{ with } \overline{n \hookrightarrow q} \longmapsto_\Delta e' \text{ with } \overline{n \hookrightarrow q}}$$

(E-Fold)
$$\frac{e \longmapsto_\Delta e'}{\mathsf{fold}_\tau\ e \longmapsto_\Delta \mathsf{fold}_\tau\ e'}$$

(E-Unfold)
$$\frac{e \longmapsto_\Delta e'}{\mathsf{unfold}_\tau\ e \longmapsto_\Delta \mathsf{unfold}_\tau\ e'}$$

Figure 2.25: Evaluation judgement for expressions

$$\boxed{\mathit{mbody}_\Delta(m, B) = e}$$

$$\textsc{(MBody-Qual)}\quad \frac{\mathit{lookup}_\Delta(q, B) = e_0}{\mathit{mbody}_\Delta(q, B, \overline{n \hookrightarrow q}) = e_0} \qquad \textsc{(MBody-Simple)}\quad \frac{\mathit{lookup}_\Delta(q, B) = e_0}{\mathit{mbody}_\Delta\big(n, B, (\ldots, n \hookrightarrow q, \ldots)\big) = e_0}$$

$$\boxed{\mathit{lookup}_\Delta(q, B) = e}$$

$$\textsc{(Lookup-Base)}\quad \frac{B(\ldots, q = e, \ldots) \text{ extends } \overline{C} \in \Delta}{\mathit{lookup}_\Delta(q, B) = e} \qquad \textsc{(Lookup-Ext)}\quad \frac{\begin{array}{c} B(\overline{q = e}) \text{ extends } \overline{C} \in \Delta \\ \text{method } q(\ldots, B.q = e, \ldots) \in \Delta \end{array}}{\mathit{lookup}_\Delta(q, B) = e}$$

$$\textsc{(Lookup-Inh)}\quad \frac{\begin{array}{c} B(\overline{q = e}) \text{ extends } \overline{C} \in \Delta \qquad q \notin \overline{q} \\ \text{method } q(\overline{D.q = e}) \in \Delta \text{ implies } B \notin \overline{D} \qquad \exists \text{ unique } k.\ \mathit{lookup}_\Delta(q, C_k) = e \end{array}}{\mathit{lookup}_\Delta(q, B) = e}$$

$$\boxed{\mathit{super}_\Delta(B \text{ as } C) = D}$$

$$\textsc{(Super-Base)}\quad \frac{B \text{ extends } D \in \Delta \qquad D \sqsubseteq_\Delta C}{\mathit{super}_\Delta(B \text{ as } C) = D} \qquad \textsc{(Super-Inh)}\quad \frac{\begin{array}{c} \nexists D' \sqsubseteq_\Delta C.\, B \text{ extends } D' \in \Delta \qquad B \text{ extends } \overline{E} \in \Delta \\ \exists D.\ \mathit{super}_\Delta(E_k \text{ as } C) = D \end{array}}{\mathit{super}_\Delta(B \text{ as } C) = D}$$

$$\boxed{\mathit{merge}\big(\overline{n_b \hookrightarrow q_b}, \overline{n_a \hookrightarrow q_a}\big) = \overline{n \hookrightarrow q}}$$

$$\mathit{merge}\big(\overline{n_b \hookrightarrow q_b}, \overline{n_a \hookrightarrow q_a}\big) = (\overline{n_b \hookrightarrow q_b}), \big\{n \hookrightarrow q \mid n \hookrightarrow q \in \overline{n_a \hookrightarrow q_a} \text{ and } n \notin \overline{n_b}\big\}$$

Figure 2.26: Auxiliary judgements for expression evaluation

2.6 Related Work

Here, using the stream examples from Sect. 2.2, Unity is compared to both mainstream languages and to closely related research languages. Other related work is presented in Sections 5.1 and 5.3. Note that for the purposes of this comparison, I assume that StringStream was defined as a sub-brand of AbstractReader (rather than AbstractWriter as in Fig. 2.4).

Java-like languages. In Java-like languages, it would be awkward to implement many aspects of the examples. First, we would have to define a number of interfaces in order to obtain the subtyping relationships displayed in Fig. 2.6 (p. 23). One plausible Java hierarchy is displayed in Fig. 2.27; compare to the Unity hierarchy in Fig. 2.28. In particular, I have defined interfaces

Reader, ResetableReader, Seekable, SeekableReader, and so on. In a real system, the situation could be even worse: here I have included the method mark in the ResetableReader interface, but some streams may support reset but *not* mark (this is true of the streams in java.io.Reader, for instance).

Additionally, since Java lacks intersection types, we must manually create compound interfaces, such as WriteableSeekable and ReadableWritableSeekable. Further, we must take care to use these interfaces rather than the interfaces they extend; for instance, StringStream implements ReadableWriteableSeekable rather than Reader, Writer, and Seekable.

There are additional issues with the Java implementation. If the type SomeOtherReader is defined in a library (independently of the Reader, etc., abstractions), we cannot write a findString method that will accept either an AbstractReader or SomeOtherReader as an argument. This is because of the lack of retroactive interface implementation: changing SomeOtherReader to implement Reader requires access to its source.

One possible workaround is to use instanceof tests:

```
boolean findString(Object o) {
   if (o instanceof AbstractReader)
      . . . // can call ‘read’, ‘skip’, ‘mark’
   else if (o instanceof SomeOtherReader)
      . . . // same code as above
   else
      throw new UnsupportedOperationException("Unknown reader");
}
```

Aside from the potential for runtime errors, there are both code duplication and extensibility problems. That is, if a new “reader” type is added (where an existing branch does not apply), programmers must remember to add a new branch to the conditional. I would like to emphasize that this is *not* a hypothetical problem: Sect. 4.5.3 provides examples of real code with this pattern.

Another workaround is to use reflection, but this solution would forgo static type safety. (One advantage of reflection over instanceof tests, however, is that the code is more extensible.)

We will also encounter a problem when trying to implement the parse method in Fig. 2.2 (p. 19). The Visitor design pattern ([Gamma et al. 1994]) is not appropriate here, since the need for it must be anticipated and it also makes class extensibility very difficult [Clifton et al. 2000].

A workaround is to again use instanceof tests, as below:

```
AST parse(ResetableReader reader) {
   if (reader instanceof CharArrayReader)
      ... // parse using ‘seek’ method
   else
      ... // use ‘mark’ and ‘reset’
}
```

Again there is an extensibility problem: if a new kind of ResetableReader is defined, the only means of extension is to update this method [Malayeri 2009c].

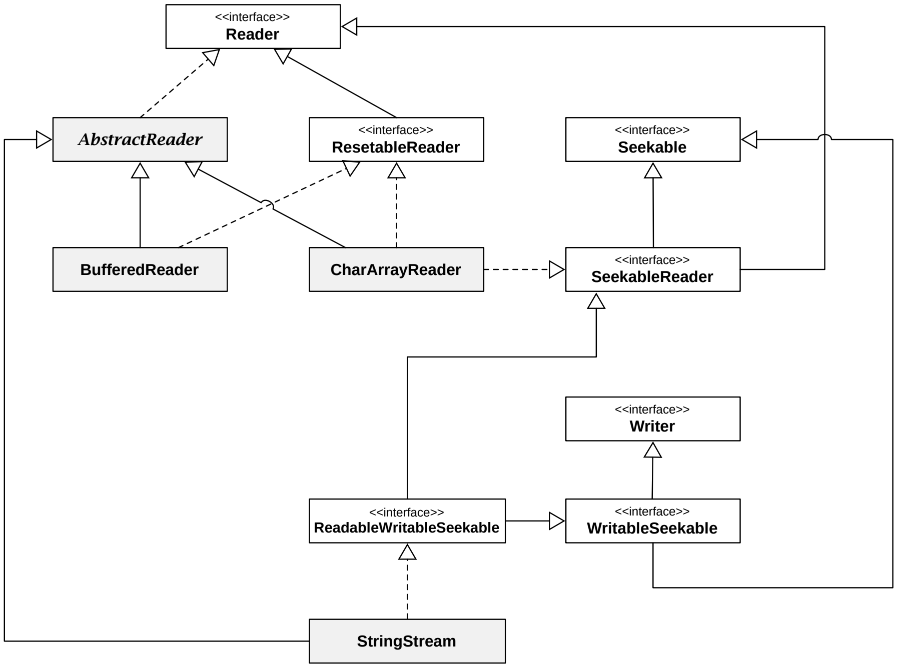

Figure 2.27: The same example implemented in Java. Classes that correspond to the Unity brands are shaded gray. Dashed lines indicate the implements relationship and solid lines indicate extends.

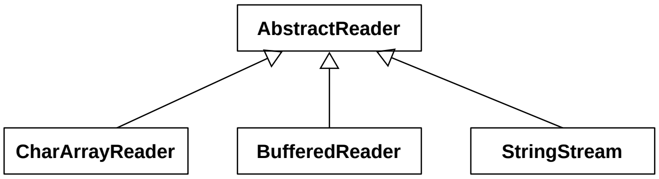

Figure 2.28: The streams example as implemented in Unity. Depicted here are the brands that must be defined in order to obtain the desired subtyping relationships.

Since seekAndWrite and readAndWrite (Fig. 2.4, p. 21) are not external methods, they can be implemented in Java using a static method that takes a WriteableSeekable and ReadableWriteableSeekable as an argument. Of course, we will encounter problems when using classes that do not implement those interfaces (as we saw with the findString method).

Finally, to implement the code in Fig. 2.5 (p. 22), a common solution is to use the Decorator pattern to implement the newline method. That is, we would define a class WriterDecorator with a field of type writer that provided an implementation of newline. (In this example, newline does not need require access to AbstractWriter methods for its implementation, but this is not the case in general.) Finally, EnhancedWriter would extend WriterDecorator and provide an implementation for writeLine.

To use the writeLine code, the method could not be called directly, as in Unity, but rather the pattern new EnhancedWriter(writer).writeLine() would have to be used. Aside from the awkwardness of this solution, there will also be problems with object identity: writer is not equal to a wrapped writer, even though they are semantically equivalent.

Overall, this example shows the impracticality and unwieldiness of implementing the Unity examples in a Java-like language.

Intersection types. In a language with nominal subtyping and intersection types [Coppo and Dezani-Ciancaglini 1978; Coppo et al. 1979; Pottinger 1980; Büchi and Weck 1998], some aspects of the Java design would be easier to express, but the same fundamental problems would remain. The benefit of intersection types is that the interfaces SeekableReader, ReadableWriteableSeekable, and WriteableSeekable would not have to be defined, since they are combinations of other interfaces.

However, the other aspects of the design would be the same as the Java design. In particular, instanceof tests would be necessary to implement findString and parse, and newline would be implemented using a decorator.

Of course, intersection types are a very useful language feature in their own right and have been included in the Unity design (see Sect. 2.5).

Functional programming languages. If we attempt to express the examples of Figures 2.1, 2.2, and 2.3 (pages 18 and 22) in an ML-like language, we would also encounter difficulties. In this case, it is due to the lack of support for typechecking extensible datatypes. In particular, there is no explicit language support for modifying a case analysis expression post-hoc—but this is precisely what would be needed to encode the Unity examples.

One approach for modeling an inheritance hierarchy in ML is to use datatypes and case analysis. Fig. 2.29 shows such an encoding. Here, we define the datatype AbstractReader with a constructor corresponding to each concrete stream class. The values carried by the constructors are simply the internal representation of each of the corresponding Unity brands. For example, the CharArrayReader constructor stores the internal character array. Methods become functions defined on AbstractReader, with a case analysis to perform different behavior for different types of streams.

However, using ML datatypes does not give us all the extensibility of Unity brands. In particular, ML-like languages have the following shortcomings:

1. Datatypes are closed to extension, unless exhaustiveness checking is sacrificed; and
2. There is no mechanism for adding new cases to existing functions without modifying them directly.

Therefore, to encode the Reader type of Fig. 2.1, we will use a record of functions. This is decidedly a more "object-oriented" style of programming and does create some awkwardness when used in ML. The new code is displayed in Fig. 2.30.

Note that the programmer must manually manage the creation of these records of functions for each subtype relation that is used. For instance, if we had a type EnhancedReader that contained the mark and reset methods, we would have to create a record to convert the CharArrayReader type to an EnhancedReader. Additionally, subtype properties such as transitivity must be encoded as functions which must be explicitly called.

```
datatype AbstractReader =
        CharArrayReader of '{' arr: char array '}'
      | BufferedReader of '{' (∗ implementation fields ∗) '}'
      | SimpleReader of '{' (∗ implementation fields ∗) '}'

fun read (r: CharArrayReader) = . . . (∗ corresponds to method CharArrayReader.read ∗)
   | read (r: BufferedReader) = . . . (∗ corresponds to method BufferedReader.read ∗)
   | read (r: SimpleReader) = . . . (∗ corresponds to method SimpleReader.read ∗)

fun skip (r: CharArrayReader) (len: int) = . . .
   | skip (r: BufferedReader) (len: int) = . . .
   | skip (r: SimpleReader) (len: int) = . . .

fun close (r: . . . ) = . . .

fun mark (r: CharArrayReader) = . . . (∗ non−exhaustive match ∗)
```

Figure 2.29: Encoding Fig. 2.1 in ML using datatypes

Up until this point, we have not encountered any major obstacles in our ML translation. However, this is no longer the case if we attempt to translate the code in Figures 2.2 and 2.3 (pages 19 and 20), as this code requires both brand extensibility and external dispatch.

Once we have encoded objects as records of functions, there is no way to write methods that externally dispatch on these objects; there is no runtime tag associated with the record. Accordingly, with this approach, it is not possible to encode the parse method of Fig 2.2 in the functional setting. If we were to change our object representation to add tags of some sort, encoding other aspects of the system would become difficult. For instance, it is unclear how to write internal methods that override external methods, as in Fig. 2.3.

As far as the author is aware, it would be impossible to encode the unique features of Unity in ML (or current extensions thereof) *and also* maintain the same static safety guarantees provided by Unity.[20] For example, Unity ensures that an internal version of a method is called, if it is more specific than an external method, and also ensures that method-not-found and method-ambiguous errors do not occur at runtime (Sect. 3.7.4). Additionally, it would not be straightforward to implement the multiple inheritance (or even single inheritance) aspects of Unity (described further in Chapter 3) in such a setting.

Finally, while there exist languages such as EML [Millstein et al. 2002; 2004], that aim to combine the strengths of functional and object-oriented languages, these languages are sufficiently different from ML that it is not at all obvious how structural subtyping would be integrated into them. Furthermore, as this dissertation is based upon the same foundations as EML, it would

[20] Certainly, it would be possible to encode Unity's dynamic semantics using ref-cells and records of functions, but this would amount to a dynamically-typed encoding. As described in Sect. 1.1, maintaining static type safety is an explicit goal of this dissertation.

```
datatype CharArrayReader = CharArrayReader of { arr: char array }
datatype BufferedReader = BufferedReader of { (∗ implementation fields ∗) }
datatype SimpleReader = SimpleReader of { (∗ implementation fields ∗) }

fun readCharArray( r: CharArrayReader ) = . . .
fun readBuffered( r: BufferedReader ) = . . .
fun readSimpleReader( r: SimpleReader ) = . . .
```

```
type 'a Reader = { read: 'a → unit → char, skip: 'a → int → int, close: 'a → unit → unit }

val charArrayFns = { read = readCharArray, skip = skipCharArray, close = closeCharArray }
val bufReaderFns = { read = readBuffered, skip = skipBuffered, close = closeBuffered }

fun findString (obj: 'a) (r: 'a Reader) (s: string) : int = . . . r#read obj . . . (∗ find the string ∗)

findString charReader charArrayFns “bar”
findString bufReader bufReaderFns “foo”

(∗ add a new type of Reader, plus conversion to Reader ∗)
datatype SomeOtherReader = SomeOtherReader of { (∗ implementation fields ∗) }
val someOtherReaderFns = . . .

findString otherReader someOtherReaderFns “baz (∗ "findString" works on new object ∗)
```

Figure 2.30: Improved ML encoding of Fig. 2.1

appear that integrating structural subtyping into that setting would pose the same challenges that are already addressed in this work.

Mixins, traits. Mixins [Bracha and Cook 1990; Ancona and Zucca 1996; Flatt et al. 1998; Findler and Flatt 1999; Ancona et al. 2003; Bettini et al. 2004] and traits [Ducasse et al. 2006; Fisher and Reppy 2004; Odersky and Zenger 2005] would allow the code to be structured somewhat differently, but the resulting design would still lack the extensibility of the Unity solution. Though there are differences between mixins and traits, these differences are not relevant for the comparison in this chapter.

To implement findString, a mixin or trait TFindStringUtil would be defined. This mixin/trait would declare the methods of Reader as required members and the findString method could therefore use these methods as in the Unity implementation. However, concrete classes that use the mixin/trait must still be defined: one for each of AbstractReader and SomeOtherReader. That is, we would have to create a class AbstractReaderFindString that inherited from AbstractReader and mixed in TFindStringUtil, and similarly for SomeOtherReader. Aside from the awkwardness of this design, the findString method could not be used directly on an object of type AbstractReader (or SomeOtherReader), one of the new subclasses must be used instead.

However, with mixins or traits, the implementation of writeLine (Fig. 2.5) would be more elegant than that of Java. We could define a mixin/trait TWriteLine that declared the methods of AbstractWriter and also the newline method as required members. We would also define a mixin/trait TNewLine with the methods of AbstractWriter as required members. Finally, we would have to create a concrete class EnhancedWriter that inherited from AbstractWriter and also mixed in TWriteLine and TNewLine. Again we would have the problem that writeLine could only be called on instances of EnhancedWriter, rather than objects of the existing types.

Sections 3.8 and 3.5.4 provide additional comparisons to mixins and traits in the context of the Unity multiple inheritance features.

Structural subtyping. Languages which support structural subtyping, such as PolyTOIL [Bruce et al. 2003], Moby [Fisher and Reppy 1999], O'Caml [Leroy et al. 2004], Scala [Odersky and Zenger 2005; Odersky 2007] and Whiteoak [Gil and Maman 2008] would elegantly express all of the desired subtyping relationships, but these languages allow only internal dispatch—that is, all methods must be defined inside the class definition. Therefore, to define a method such as parse (Fig. 2.2), as in Java, some form of instanceof test must be used.

Of the aforementioned languages, only Scala allows a type to have both a nominal and structural component, which is achieved using intersection types. However, Scala does not allow defining recursive structural types (such as Writer in Fig. 2.4). Consequently, Writer would have to be a nominal type, and we would once again encounter the problems of retroactive interface extension.

Additionally, these languages do not allow structural constraints to be placed on a method's receiver (such as the parse method in Fig. 2.2). While this is consistent with the single dispatch semantics that gives special status to a method's receiver, in a external or multiple dispatch setting, the Unity design is more uniform.

ML-style and first-class modules. ML-style modules [Milner et al. 1997] use structural signature matching, and as such could be used to encode the subtype polymorphism aspect of a structurally-typed object-oriented language. Such an encoding would not be as expressive as Unity, however, as one would need the capability of dynamically selecting among types defined in different modules. With ordinary ML modules, this would require a global re-write of the program. Additionally, this encoding would likely require a large degree of functorization.

A first-class module system (e.g., [Dreyer 2005; Dreyer et al. 2003; Dreyer and Rossberg 2008]) would solve these problems, but such designs do not directly address the problems considered in this dissertation (though they are quite useful in their own right). In particular, the analogue to external dispatch in this setting is unclear: even first-class modules do not provide a tagging (let alone sub-tagging) mechanism, so one would have to fall back on the datatype encoding from above, which was shown to lack the desired properties.

Retroactive interface extension. Some languages, such as JavaGI [Wehr et al. 2007] and Cecil [Chambers and the Cecil Group 2004] provide both external/multimethod dispatch and *retroactive interface extension*. That is, after a class has been defined, new external methods can be defined for it, or it can be declared to extend a new interface, or both. However, Cecil [Chambers and the Cecil Group 2004] requires whole-program typechecking. JavaGI [Wehr et al. 2007] performs some modular typechecking, but defers some type checks to class-load time, even when dynamic classloading is not used.

In fact, there appears to be a fundamental tension between retroactive (nominal) interface extension and modular compilation and typechecking. The situation is exacerbated with tagged interfaces; such a feature would essentially allow adding new tags to existing objects. This poses problems both for typechecking external methods and for program reasoning. The details of these problems are described in Sect. 5.1.

One compromise is to use *expanders*, which allow typesafe, statically-scoped object adaptation [Warth et al. 2006]. Using expanders, programmers can non-invasively update existing classes to add new fields and methods and to implement new interfaces. The statically scoped nature of expanders is particularly appealing, as clients have flexible control over the visibility of the adapted classes. However, expanders are not as flexible as full structural subtyping. Though it is possible to use "expander overriding" to extend existing expanders, some forms of post-hoc changes are still difficult with expanders. For instance, if one expander E_1 augments class C with method m, it is not possible for another expander E_2 to extend E_1 and make C now implement interface I (which requires method m). Accordingly, expanders are only a partial solution to the problem of retroactive abstraction.

Scala "implicits" (formerly called *views*), allow programmers to easily define adapters from one object to another [Odersky 2007] and are very useful as a workaround for the lack of retroactive abstraction in Java-like languages. Among other things, implicits allow programmers to specify specify method renamings. As implicits are lexically scoped, they are similar to Unity's "using ... in" expression (Sect. 2.4.3).

Type classes. Many aspects of the Unity examples can be quite elegantly encoded using type classes, which provide a principled form of ad-hoc polymorphism [Wadler and Blott 1989; Hall et al. 1996]. However, since type classes have no notion of subtyping, the encoding would lack some of Unity's expressiveness.

Fig. 2.1 can be readily encoded using type classes; Reader would be defined as a type class and the programmer would provide "instance" declarations to make CharArrayReader, BufferedReader and SomeOtherReader members of the Reader type class. Instance declarations can be defined after a type has been defined, so in that regard they provide a form of retroactive abstraction.

The difference between the type class encoding and the Unity version is more apparent for the listing in Fig. 2.4. StringStream does not have to be declared as an instance of the type class consisting of the seek method, for instance—once all the method definitions have been provided, structural subtyping provides the necessary coercions. Since type classes do not provide a hierarchy of any form, programmers would have to manually provide coercion functions to convert from one type class to another. For example, suppose we have some function f defined for values of the Equality type class (which contains an equals function). If integer is a member of Comparable (which has both compares and equals functions), then one would need a comparableToEquality coercion to pass an integer to f. Similarly, while in Unity we can define an

```
forall MapType, SetType, CollectType, SetIterType, CollectIterType:
    method useMap(m: MapType): void
        where signature entrySet(MapType): SetType and
        signature values(MapType): CollectType and

        SetType <= SetBrand and
        signature contains(SetType, Object): bool and
        signature iterator(SetType): SetIterType and
        . . . -- additional Set methods

        signature contains(CollectType, Object): bool and
        signature iterator(CollectType): CollectIterType and
        . . . -- additional Collection methods

        signature next(SetIterType): Object and
        signature hasNext(SetIterType): Object and

        -- exact same lines as above
        signature next(CollectIterType): Object and
        signature hasNext(CollectIterType): Object
{. . .}
```

Figure 2.31: Translating the structural types of Fig. 2.8 to Cecil "where" clause constraints.

external method and automatically widen the interface of the class on which it is defined, with type classes we would have to provide a coercion manually.

Additionally, type classes do not provide any form of hierarchical dispatch, so an additional language mechanism would be necessary for this functionality.

Note that type classes are often implemented in a manner similar to Unity's simple-to-qualified map. In particular, if a Haskell function f is defined for types that are members of a particular type class C, then f is compiled to a function that takes an additional argument: a dictionary implementing the methods of C [Hall et al. 1996]. This dictionary is similar in flavor to the Unity name map that is carried along with each object value.

Cecil. Cecil fully supports external and multimethod dispatch [Chambers 1992; Chambers and the Cecil Group 2004] and can also be used to encode some structural subtyping patterns using constraint-bounded polymorphism ("where" clauses). Cecil's powerful, but very complex, type system can express most of the necessary relationships in my Collections example, but the type constraints can be come verbose.

In particular, to translate the writeLine function (Fig. 2.5), a programmer would use bounded quantification and a "where" clause constraint, the latter being typechecked via a constraint solver [Chambers and the Cecil Group 2004; Litvinov 2003]. That is, in psuedo-code, the argument to writeLine would have type:

```
forall I: constraint Iterator =  -- fictional abbreviation for a where clause constraint
    signature next(I): Object and
    signature hasNext(I): bool

forall C, I: constraint Collection =
    signature contains(C, Object): bool and
    signature iterator(C): I and
    Iterator [I]
. . .

forall  MapType, SetType, CollectType, SetIterType, CollectIterType:
    method useMap(m: MapType): void
        where signature entrySet(MapType): SetType and
        signature values(MapType): CollectType and
        Set [SetType, SetIterType] and Collection [CollectType, CollectIterType]
```

Figure 2.32: A possible syntactic sugar for the Cecil "where" clause constraints of Fig. 2.31. Note that even with this syntax, useMap still needs five type parameters

```
forall T: method writeLine(t: T): void
    where T <= AbstractReader and signature mark(T): void and signature reset(T): void
```

Here there is not an excessive amount of user-specified constraints, but the problem is compounded when translating nested structural types to "where" clauses.

For example, to achieve the expressiveness of the Unity code in Fig. 2.8, the method useMap would require 5 type parameters and over 16 "where" constraints. A subset of these constraints is shown in Fig. 2.31. Note the repetition in the constraints for the pair SetType and CollectType and the pair SetIterType and CollectIterType. A distinct type parameter is needed for encoding each nested structural type, since "where" clauses can only be specified on type parameters.

Of course, it would be possible to define syntactic sugar to simplify the definition of such a method. For instance, the language could provide constraint abbreviations, such as those illustrated in Fig. 2.32.

However, there is still the overhead of having to add a type parameters wherever a where clause is needed. As a consequence, every method call from the point where the type parameter is instantiated to where the object is finally used, must also be parameterized and include the constraint. That is, if we wish to write a function foo that calls useMap, it must either instantiate the 5 type parameters or must itself be parameterized. This latter design would be necessary for maximal reusability of foo. Essentially, requiring the use of parametric polymorphism means that everything must be parameterized "all the way up" to the point where actual parameters are provided.

Additionally, since only classes and methods may have type parameters, if we wish use a structural type in the body of a method, we must added a corresponding type parameter to the method.

In contrast, in Unity, structural types are compositional (allowing them to be nested within other types) and can occur at any level in the program (e.g., the first-class functions findString and seekAndWrite).

It should be noted, however, that parameterization can be used to specify more precise types, since a subtype may only be substituted when explicitly allowed by the constraint (i.e., type parameters are invariant by default). By using type parameters, programmers may also specify a relationship between type parameters, such as with the Subject/Observer design pattern [Litvinov 2003].

In addition, I have developed a version of Unity with polymorphism [Malayeri and Aldrich 2008b] and it does not appear that adding (ordinary) bounded quantification would pose any noteworthy complications. In Unity, additional type parameters would be necessary only when this extra precision is needed. Thus, *in the context of the features of this chapter*, I argue that Unity is strictly less verbose and more expressive than the corresponding Cecil encoding.

2.7 Conclusion

This chapter showed how a combination of nominal and structural subtyping has a number of benefits, the most important of which is that it allows structural subtyping and external methods to gracefully co-exist. I also showed that there is a synergy between these two features: structural subtyping can be used to define an interface to which brands can retroactively conform. Then, in the case where additional code is needed to provide this conformance, external methods can be used. Thus, I have provided evidence that supports hypotheses I and IV.

Chapter 3

Multiple Inheritance

> I see young men, my townsmen, whose misfortune it is to have inherited farms, houses, cattle, barns, and farming tools, for these are more easily acquired than gotten rid of.
>
> Henry David Thoreau (*Walden*)

This chapter details the multiple inheritance feature of Unity,[1] starting with an overview of the problem and previous solutions. I then present the details of the multiple inheritance design (including the formalization) and provide an extended example and two real-world examples.

3.1 Overview

Single inheritance, mixins [Bracha and Cook 1990; Flatt et al. 1998], and traits [Ducasse et al. 2006; Odersky and Zenger 2005] each have disadvantages: single inheritance restricts expressiveness, mixins must be linearly applied, and traits do not allow state. Multiple inheritance is one solution to these problems, as it allows code to be reused along multiple dimensions. Unfortunately however, multiple inheritance poses challenges itself.

As mentioned in Sect. 1.3, there are two well-known problems with multiple inheritance: (a) a class can inherit multiple features with the same name, and (b) a class can have more than one path to a given ancestor, i.e., "diamond inheritance" [Sakkinen 1989; Singh 1994]. As the first problem can be solved by allowing renaming (e.g., Eiffel [Meyer 1997]) or by linearizing the class hierarchy [Snyder 1986; Singh 1994], I shall focus on problems caused by diamond inheritance.

Recall that diamond inheritance occurs when a class (or brand) C inherits an ancestor A through more than one path. This is particularly problematic when A has fields—should C inherit multiple copies of the fields or just one? Virtual inheritance in C++ is designed as one solution for C to inherit only one copy of A's fields [Ellis and Stroustrup 1990]. But with only

[1] The primary technical contributions of this chapter appear in [Malayeri 2009a;b; Malayeri and Aldrich 2009a].

one copy of *A*'s fields, object initializers are a problem: if *C* transitively calls *A*'s initializer, how can we ensure that it is called only once? As we shall see in Sect. 3.3, existing solutions either lack expressiveness or can cause semantic problems.

More importantly, however, diamond inheritances causes multiple inheritance to interact poorly with modular typechecking of external dispatch. In fact, *any form of multiple inheritance*—even Java-style multiple interface inheritance—suffers from this problem. Previous work has not satisfactorily addressed this problem: existing solutions (described in Sect. 3.3) either restrict expressiveness or require an exponential number of programmer-supplied disambiguating methods.

Unity takes a novel approach to this problem: while permitting multiple inheritance, it forbids inheritance diamonds. To ensure no loss of expressiveness, the notion of inheritance is divided into two concepts: an *inheritance dependency* (expressed using a requires clause, an extension of a Scala construct [Odersky and Zenger 2005; Odersky 2007]) and ordinary inheritance. Through examples, this chapter illustrates how Unity retains the expressiveness of diamond inheritance: programs that require diamond inheritance can be systematically translated to a hierarchy that uses a combination of requires and no-diamond multiple inheritance.

Additionally, I claim that a hierarchy with multiple inheritance is conceptually two or more separate hierarchies. These hierarchies represent different "dimensions" of the brand that is multiply inherited. Dependencies between these dimensions are expressed using requires; Sect. 3.5 presents an extended example of this technique.

The solution has two advantages: fields and multiple inheritance (including initializers) can gracefully co-exist, and external/multiple dispatch and multiple inheritance can be combined. To achieve the latter, I have made an incremental extension to existing techniques for modular typechecking of external and multiple dispatch.[2]

An additional feature of the language is a dynamically-dispatched super call, modelled after trait super calls [Ducasse et al. 2006]. When a call is made to $A.\mathsf{super}.f()$ on an object with dynamic type *D*, the call proceeds to *f* defined within *D*'s immediate super-brand along the *A* path (i.e., some *E* where *D* extends *E* and $E \sqsubseteq A$). The brand *A* must be specified because *D* may have more than one parent. With dynamically-dispatched super calls and requires, Unity attains the expressiveness of traits while still allowing brands to inherit state.[3]

3.2 The Problem

To start with, diamond inheritance raises a question of semantics: should class (or brand) *C* with a repeated ancestor *A* have two copies of *A*'s instance variables or just one—i.e., do we wish to have *tree* or *graph* [Carré and Geib 1990] inheritance semantics? As the former may be modelled using composition and (method call) forwarding, the latter is the desirable semantics, a semantics that is supported by Scala, Eiffel, and C++ virtual inheritance [Odersky 2007;

[2] Without loss of generality, recall that my formal system includes external methods rather than full multimethods; multimethods can be encoded using external methods (Sect. 2.4.2).

[3] It would also be possible to add additional features present in traits, such as the "exclude" and "alias" operators. These features will not be discussed further in this dissertation, as they are orthogonal to the problems on which I have focused.

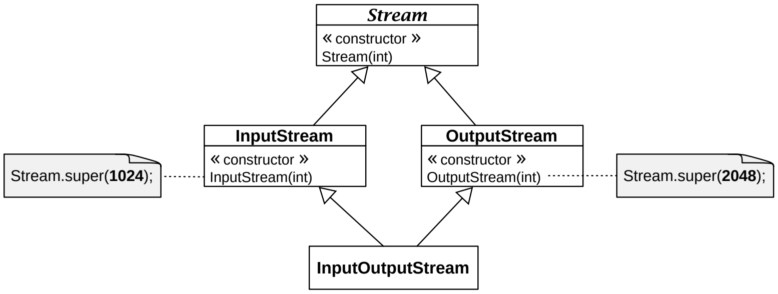

Figure 3.1: Stream classes in an inheritance diamond. Italicized class names indicate abstract classes.

Meyer 1997; Ellis and Stroustrup 1990]. Unity provides *only* graph inheritance semantics—an intentional design choice.

Next, diamond inheritance leads to (at least) two major problems that have not been adequately solved: (1) determining how and when the superclass constructor/initializer should be called [Snyder 1986; Singh 1994], and (2) how to ensure non-ambiguity of multimethods in a modular fashion [Millstein and Chambers 2002; Frost and Millstein 2006; Allen et al. 2007]. Note that the first problem only arises with graph inheritance semantics, while the second occurs in either semantics (tree or graph).

Object Initialization

To illustrate the first problem, consider Figure 3.1, which shows a class hierarchy containing a diamond. Suppose that the Stream superclass has a constructor taking an integer, to set the size of a buffer. InputStream and OutputStream call this constructor with different values (1024 and 2048, respectively). But, when creating an InputOutputStream, with which value should the Stream constructor be called? Moreover, InputStream and OutputStream could even call different constructors (with different parameter types), making the situation even more uncertain.

Modular Multiple Dispatch

The second problem regards multiple dispatch, which I and others believe is more natural and expressive than single dispatch [Chambers 1992; Clifton et al. 2000; Chambers and the Cecil Group 2004]. However, typechecking multiple dispatch in a modular fashion becomes very difficult in the presence of multiple inheritance—precisely because of the diamond problem.

As previously noted, I focus on external methods (also known as open classes), which are essentially multimethods that dispatch on the first argument only (corresponding to the receiver of an ordinary method). As previously described in Section 2.4.2, multimethods with asymmetric dispatching semantics (where the order of arguments affects dispatch) can be translated to external methods in a straightforward manner. Thus, my language proposal is applicable to languages with multimethods as well.[4]

[4] It happens that the Unity solution is applicable to languages with either asymmetric or symmetric dis-

To see why diamond inheritance causes problems, consider the following definition of the external method seek (defined externally for illustrative purposes):

```
method Stream.seek (
    Stream.seek: ( ) ⇒ long → unit = fn _ --> ( )  // default implementation: do nothing
    InputStream.seek: ( ) ⇒ long → unit = . . .  // seek if first arg <= eofPos
    OutputStream.seek: ( ) ⇒ long → unit = . . .  // if first arg > eofPos, fill with zeros
)

inOutStream.seek 10 // ambiguous!
```

In the context of our diamond hierarchy, this method definition is ambiguous—what if seek( ) is called on an object of type InputOutputStream? Unfortunately, it is difficult to perform a *modular* check to determine this fact. When typechecking the definition of seek( ), we cannot search for a potential sub-brand of both InputStream and OutputStream, as this analysis would not be modular. And, when typechecking InputOutputStream, we cannot search for external methods defined on both of its super-brands, as that check would not be modular, either. A detailed description of the conditions for modularity is provided in Sect. 3.7.3.

Note that this problem is *not* confined to multiple (implementation) inheritance—it arises in any scenario where an object can have multiple dynamic types (or tags) on which dispatch is performed. For instance, the problem appears if dispatch is permitted on Java interfaces, as in JPred [Frost and Millstein 2006].

3.3 Previous Solutions

Here, I describe previous solutions that specifically address the object initialization and modular multiple dispatch problems. Additional related work is described in Sect. 3.8.

Object Initialization

Languages that attempt to solve the object initialization problem include Eiffel [Meyer 1997], C++ [Ellis and Stroustrup 1990], Scala [Odersky 2007] and Smalltalk with stateful traits [Bergel et al. 2008].

In Eiffel, even though (by default) only one instance of a repeatedly inherited class is included (e.g., Stream), when constructing an InputOutputStream, the Stream constructor is called twice. This has the advantage of simplicity, but unfortunately it does not provide the proper semantics; Stream's constructor may perform a stateful operation (e.g., allocating a buffer), and this operation would occur twice.

In C++, if virtual inheritance is used (so that there is only one copy of Stream), the constructor problem is solved as follows: the calls to the Stream constructor from InputStream and OutputStream are ignored, and InputOutputStream must call the Stream constructor explicitly. (Since there is no default Stream constructor, this call cannot be automatically generated.) Though the Stream constructor is called only once, this awkward design has the problem that

patch semantics—the latter introduces a few orthogonal typechecking issues. Millstein and others provide a detailed discussion of the topic [Millstein et al. 2002; Millstein 2003].

constructor calls are ignored. The semantics of InputStream may require that the Stream fields be constructed in a particular manner, but C++ effectively ignores this dependency.

Scala provides a different solution: trait constructors may not take arguments. (Scala traits are abstract classes that may contain state and may be multiply inherited.) This ensures that InputStream and OutputStream call the same super-trait constructor, causing no ambiguity for InputOutputStream. Though this design is simple and elegant, it restricts expressiveness. (In fact, for the next major release, the Scala designers wish to attain a solution that permits trait constructor arguments [Washburn 2008].)

Smalltalk with stateful traits [Bergel et al. 2008] does not contain constructors, but by convention, objects are initialized using an initialize message. Unfortunately, this results in the same semantics as Eiffel; here, the Stream constructor would be called twice [Bergel 2008]. The only way to avoid this problem would be to always define a special initializer that does not call the superclass initializer. Requiring that the programmer define such a method essentially means that the C++ solution must be hand-coded. Aside from being tedious and error-prone, this has the same drawbacks as the C++ semantics.

Mixins and (stateless) traits do not address the object initialization problem directly, but instead restrict the language so that the problem does not arise in the first place. I compare Unity to each of these designs in Sect. 3.8.

Modular Multiple Dispatch

There are two main solutions to the problem of modular typechecking of multiple dispatch (or external methods) in the presence of multiple inheritance. The first solution is to simply disallow multiple inheritance across module boundaries; this is the approach taken by the "System M" variant of Dubious [Millstein and Chambers 2002]. Obviously, the disadvantage here is the loss of expressiveness.

JPred [Frost and Millstein 2006] and Fortress [Allen et al. 2007] take a different approach. The diamond problem arises in these languages due to multiple interface inheritance and multiple trait inheritance, respectively. In these languages, the typechecker ensures that external methods are unambiguous by requiring that the programmer always specify a method for the case that an object is a subtype of two or more incomparable interfaces (or traits). In our streams example, the programmer would have to provide a method like the following (in JPred syntax):

```
void seek(Stream s, long pos) when s@InputStream && s@OutputStream { }
```

(In Fortress, the method would be specified using intersection types.) Note that in both languages, this method would have to be defined for *every* subset of incomparable types, regardless of whether a type like InputOutputStream is ever defined. That is, even if two types will *never* have a common subtype, the programmer must specify a disambiguating method, one that perhaps throws an exception.[5] Thus, the problem with this approach is that the programmer is required to write numerous additional methods—exponential in the number of incomparable

[5] In Fortress, the programmer may specify that two traits are disjoint, meaning that there will never be a subtype of both. To allow modular typechecking, this disjoint specification must appear on one of the two trait definitions, which means that one must have knowledge of the other; consequently, this solution lacks sufficient extensibility and scalability.

types—some of which may never be called. JPred alleviates the problem somewhat by providing syntax to specify that a particular branch should be preferred in the case of an ambiguity, but it may not always be possible for programmers to know in advance which method should be preferred.

Neither JPred interfaces nor Fortress traits may contain state and thus the languages do not provide a solution to the object initialization problem; the language Dubious does not either, as it does not contain constructors.

3.4 Multiple Inheritance in Unity

This section first gives an informal description of Unity's multiple inheritance features, along with an illustrative example. I show how Unity solves our two main problems (object initialization and modularly typchecking external methods) and then provide a more detailed account of the typechecking rules. This section also provides additional comparison to the relevant related work.

3.4.1 Overview

Unity's multiple inheritance design is based on the intuition that there are semantic relationships between brands that are not captured by inheritance, and that if brand hierarchies could express richer interconnections, inheritance diamonds need not exist. Suppose the concrete brand C extends A. As noted by Schärli et al. [2003], it is beneficial to recognize that C serves two roles: (1) it is a generator of instances; and (2) it is a unit of reuse, through sub-branding. In the first role, inheritance is the implementation strategy and must be preserved (assuming extensive redesign is not feasible). In the second role, however, it is possible to transform the brand hierarchy to one where an inheritance *dependency* between C and A is declared and where *sub-brands* of C inherit from both C and A. This notion of inheritance dependency is the key distinguishing feature of multiple inheritance in Unity: while multiple inheritance is permitted, inheritance diamonds are forbidden.

Consider the inheritance diamond of Fig. 3.1. To translate this hierarchy to Unity, InputStream would be made abstract[6] and its relationship to Stream would be changed from inheritance to an inheritance *dependency*, which means that (concrete) sub-brands of InputStream must also inherit from Stream. In other words, InputStream *requires the presence of* Stream *in the* extends *clause of concrete sub-brands*, but it need not extend Stream itself. Since InputStream is now abstract (making it serve only as a unit of reuse), it can be safely treated as a subtype of Stream. However, any concrete sub-brands of InputStream (generators of instances), must also inherit from Stream. Accordingly, InputOutputStream must inherit from Stream directly.

This notion of an inheritance dependency is reified using the requires keyword, a generalized form of a similar construct in Scala [Odersky and Zenger 2005; Odersky 2007].[7]

[6] While the calculus does not have a formal notion of "abstract" and "concrete," it ensures that conceptually "abstract" brands may not be instantiated.

[7] In Scala, requires is used to specify the type of a method's receiver (i.e., it is a selftype), and does not create a subtype relationship. As far as the Scala team is aware, our proposed use of requires is novel [Washburn

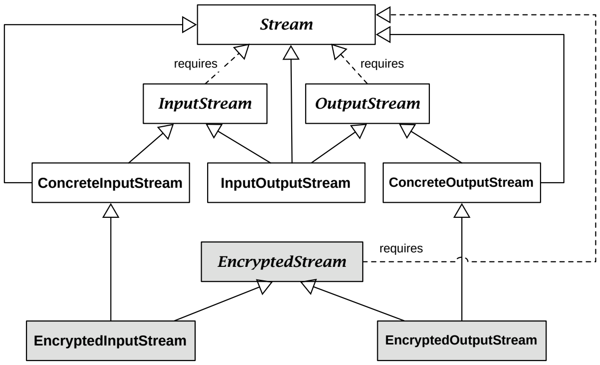

Figure 3.2: The stream hierarchy of Fig. 3.1, translated to Unity, with an encryption extension in gray. Italicized brand names indicate abstract brands, solid lines indicate extends, and dashed lines indicate requires.

Definition 3.1 (Requires).
When a brand C requires a brand B, we have the following:

1. C objects may not be created (i.e., C is abstract)
2. C objects are subtypes of B objects ($C() \leq B()$), but C is not a sub-brand of B ($C \not\trianglelefteq B$)
3. Sub-brands of C must either require B themselves (making them abstract) or extend B (allowing them to be concrete).[8] This is achieved by including a requires B' or extends B' clause, where B' is a sub-brand of B.

In essence, "C requires B" defers the actual inheritance of B (i.e., sub-branding), but provides a guarantee that C's concrete sub-brands will extend B (or one of its sub-brands).

The revised stream hierarchy is displayed in Fig. 3.2. In the original hierarchy, InputStream served as both generator of instances and a unit of reuse. In the revised hierarchy, we divide the brand in two—one for each role. The brand ConcreteInputStream is the generator of instances, and the abstract brand InputStream is the unit of reuse. Accordingly, InputStream requires Stream, and ConcreteInputStream extends both InputStream and Stream. The concrete brand InputOutputStream extends each of Stream, InputStream, and OutputStream, creating a sub*typing* diamond, but not a sub-*branding* diamond.

The code for InputStream will be essentially the same as before, except for the call to its super constructor (explained further below). Because InputStream is a subtype of Stream, it may use all the fields and methods of Stream, without having to define them itself.

Programmers may add another dimension of stream behavior through additional abstract brands, for instance EncryptedStream. EncryptedStream is a type of stream, but it need not extend Stream, merely require it. Concrete sub-brands, such as EncryptedInputStream must inherit from

2008].

[8] This propagation of the requires clause is not strictly necessary and could be inferred; however, it is included by analogy with Scala as well as to simplify the calculus.

Stream, which is achieved by extending ConcreteInputStream. (It would also be possible to extend Stream and InputStream directly.)

The requires relationship can also be viewed as declaring a semantic "mixin"—if B requires A, then B is effectively stating that it is an extension of A that can be "mixed-in" to clients. For example, EncryptedStream is enhancing Stream by adding encryption. Because the relationship is explicitly stated, it allows B to be substitutable for A.

Using requires is preferable to using extends because the two brands are more loosely coupled. For example, we could modify EncryptedInputStream to require InputStream (rather than extend ConcreteInputStream). A concrete sub-brand of EncryptedInputStream could then also extend a *sub-brand* of InputStream, such as BufferedInputStream, rather than extending InputStream directly. In this way, different pieces of functionality can be combined in a flexible manner while avoiding the complexity introduced by inheritance diamonds.

Object Initialization

Because there are no inheritance diamonds, the object initialization problem is trivially solved. Note that if brand C requires A, it need not (and should not) call A's constructor, since C does not inherit from A.[9] In our example, InputStream does not call the Stream constructor, while ConcreteInputStream calls the constructors of its super-brands, InputStream and Stream. Thus, a sub*typing* diamond does not cause problems for object initialization.

This may seem similar to the C++ solution; after all, in both designs, InputOutputStream calls the Stream constructor. However, the Unity design is preferable for two reasons: a) there are no constructor calls to non-direct super-brands, and, more importantly, b) constructor calls are never ignored. In the C++ solution, InputStream may expect a particular Stream constructor to be called; as a result, it may not be properly initialized when this call is omitted. Essentially, Unity does not allow the programmer to create constructor dependencies that cannot be enforced.

Modular Multiple Dispatch

A similar principle solves the problem of modular multiple dispatch. In Unity, a method may only override a method in a super-brand, not a required brand.[10] So, the definitions of InputStream.seek and OutputStream.seek cannot not override Stream.seek—methods defined in an external method block must be a sub-brand of the method's owner brand.

Let us suppose for a moment that all brands in Fig. 3.2 have been defined, except InputOutputStream. Accordingly, we would re-write the seek methods as in Fig. 3.3. (Though these definitions are slightly more verbose than before, syntactic sugar could be provided.) Note that seekInput and seekOutput could just as easily have been ordinary functions (lambda expressions) in this example, as there is no overriding here.

[9] While the formal system does not include constructors (fields are simply initialized directly when an object is created), this would be quite a straightforward extension of the system.

[10] Note that it would be possible to remove this restriction for internal methods, as any ambiguity is easily detected modularly. However, such a semantics unduly complicates the formal system and does not add any expressiveness. This is due to the fact that Unity does not define any sort of linearization, so programmers must define this manually anyway.

```
// helper methods, could also use lambda expressions
method InputStream.seekInput: ( ) ⇒ long → unit = . . .
method OutputStream.seekOutput : ( ) ⇒ long → unit = . . .

method Stream.seek (
    Stream.seek: ( ) ⇒ long → unit = fn _ --> ( )  // default implementation: do nothing
    ConcreteInputStream.seek: ( ) ⇒ long → unit =
        fn pos: long --> this.seekInput pos
    ConcreteOutputStream.seek: ( ) ⇒ long → unit =
        fn pos: long --> this.seekOutput pos
)
```

Figure 3.3: Re-writing the seek method in Unity.

Note that the typechecker does *not* require that a disambiguating method be provided for "InputStream && OutputStream", unlike JPred and Fortress. If a programmer later defines InputOutputStream, but does not re-define seek, the default implementation of Stream.seek will be inherited. An external or internal method for InputOutputStream can then be implemented, perhaps one that calls OutputStream.doSeek( ).

Here, it is of key importance that sub-brand diamonds are disallowed; because they cannot occur, external methods can be easily checked for ambiguities. Sub*typing* diamonds do not cause problems, as external method overriding is based on sub-*branding*.

Using requires

Introducing two kinds of brand relationships raises the question: when should programmers use requires, rather than extends? A rule of thumb is that requires should be used when a brand is an extension of another brand and is itself a unit of reuse. If necessary, a concrete brand extending the required brand (such as ConcreteInputStream) could also be defined to allow object creation. Note that this concrete brand definition would be trivial, likely containing only a constructor. On the other hand, when a brand hierarchy contains multiple disjoint alternatives (such as in the AST example in the next section), extends should be used; the no-diamond property is also a semantic property of the brand hierarchy in question.

The above guideline may result in programmers defining more abstract brands (and corresponding concrete brands) than they may have otherwise used. However, some argue that it is good design to make a class (or brand) abstract whenever it can be a base class (or brand). This is in accordance with the design of classes in Sather [Szyperski et al. 1993], traits in Scala and Fortress [Odersky 2007; Allen et al. 2008; 2007], and the advice that "non-leaf" classes in C++ be abstract [Meyers 1992]. In Sather and Fortress, for example, only abstract classes may have descendants; concrete classes (called "objects" in Fortress) form the leaves of the inheritance hierarchy [Szyperski et al. 1993]. Futhermore, a language could define syntactic sugar to ease the task of creating concrete brand definitions; such a design is sketched in Sect. 3.5.4.

3.4.2 Typechecking Multiple Inheritance

Here, I describe the multiple inheritance typechecking rules at a high level, to provide an intuition as to why typechecking is modular. Sect. 3.7 completes the discussion of Unity's formalism (i.e., the highlighted portions of Sect. 2.5) and also provides a detailed argument of its modularity.

Brands

As mentioned, inheritance diamonds are forbidden in Unity. Concretely, we have the following condition:

***B1*.** If a brand B extends C_1 and C_2 then there *must not* exist some D, other than Object, such that both C_1 and C_2 are sub-brands of D.

A special case is made for the brand Object—the root of the inheritance hierarchy, since every brand directly or indirectly extends it. (Otherwise, a brand could never extend two unrelated brands—the existence of Object would create a diamond.) Note that this does not result in the object initialization problem, because Object has only a no-argument constructor. Also, this condition does not preclude a brand from inheriting from two concrete brands if this does not form a diamond.

Additionally, our convention of unique method name introductions (Sect. 2.4.3) ensures that ambiguities cannot arise when two unrelated brands A and B coincidentally have the same name and a third brand inherits from both A and B.[11]

External Methods

The restrictions on external methods, conditions *E1*–*E3*, were enumerated in Sect. 2.4.2. While all three conditions are the same as those in System M, that language did not allow multiple inheritance across module boundaries. In Unity this restriction is removed by ensuring that diamond inheritance does not occur—condition *B1*. (Note that in Unity, each brand and each top-level method declaration is in its own "module.")

Recall that the rationale for conditions *E1* and *E2* were described in Sect. 2.4.2, but the discussion of condition *E3* was deferred. This condition is imposed for modular ambiguity checking in the presence of multiple inheritance:

***E3*.** When an external method family m is introduced, it must declare an *owner brand* C: this specifies that the method family is rooted at C. C must be a proper subtype of Object, the root of the inheritance hierarchy. An external method definition m for brand D is valid only if D is a sub-*brand* of C.

Condition *E3* ensures that diamonds with Object at the top (permitted by condition *B1*) do not cause an ambiguity. Concretely, consider the following method definition:

[11] Incidentally, this is not the convention used in Java interfaces, but is that of C#.

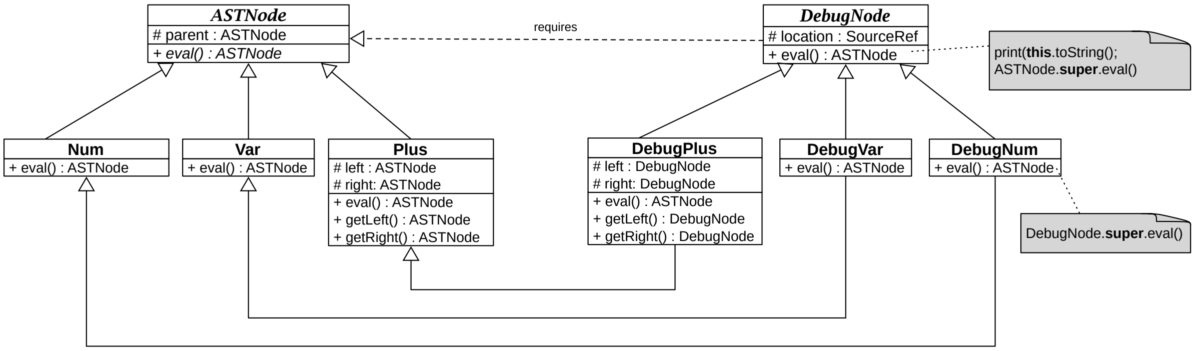

Figure 3.4: The AST node example in Unity. Abstract brands and abstract methods are set in italic. The visibility modifiers '+', '-' and '#' indicate public, private and protected, respectively.

```
method Object.g (  // illegal definition–owner cannot be Object
    Stream.g : ( ) ⇒ unit = . . .
    Foo.g : ( ) ⇒ unit = . . .
)
brand Bar extends Stream, Foo ( . . . )  // problem! two versions of g( )!
```

If this were valid code, there would exist a method definition g( ) for each of Stream and Foo. In this case, Bar would inherit two equally legitimate definitions of g( ). For typechecking to be modular, when checking Bar, we should not have to check all definitions of external methods, including g( ). Note that *not* specifying an owner brand would have the same effect as using Object as an owner.

Additionally, condition *E3* ensures that diamond sub*typing* (as opposed to sub-branding) does not result in a brand inheriting the same external method through more than one path. If overriding were permitted based on subtyping, the problem described with diamond inheritance (Sect. 3.2) would re-appear.

The owner brand is also useful for implementing unique qualified names, described in Sect. 2.4.3. (A related issue, defining two external methods with the same name *m*, can be resolved by using a naming convention for modules.)

3.5 Example: Abstract Syntax Trees

Consider a simple type hierarchy for manipulating abstract syntax trees (ASTs), such as the one in Fig. 3.4. The original hierarchy is the one on the left, which consists of ASTNode, Num, Var, and Plus. An ASTNode contains a reference pointing to its parent node, as indicated in the figure. Each of the concrete sub-brands of ASTNode implements its own version of the abstract ASTNode.eval( ) method. For the sake of brevity, methods for accessing the state of Num and Var have been omitted.

Suppose that after we have defined these brands, we wish to add a new method that operates over the AST. For instance, we may want to check that variables are declared before they are used (assuming a variable declaration statement). Since Unity supports external methods, a method defCheck( ) could be added externally as follows:

```
method ASTNode.defCheck (  // external method
    ASTNode.defCheck : ( ) ⇒ bool = . . .
    Num.defCheck : ( ) ⇒ bool = . . .
    Var.defCheck : ( ) ⇒ bool = . . .
    Plus.defCheck : ( ) ⇒ bool = . . .
)
```

Note that the programmer would *only* have to define cases for Num, Var and Plus; she need not specify what method should be called when an object has a combination of these types—such a situation cannot occur (as there are no diamonds).

Now, suppose we wish to add debugging support to our AST, after the original hierarchy is defined. Each node now additionally has a source location field, DebugNode.location. Debugging support, on the right side of the figure, is essentially a new dimension of AST nodes that has a dependency on ASTNode. We express this using requires. Now, brands like DebugPlus can multiply inherit from ASTNode and DebugNode without creating a sub-branding diamond. In particular, DebugPlus does *not* inherit two copies of the parent field, because DebugNode is a subtype, but not a sub-brand, of ASTNode. Thus, the no-diamond property allows fields and multiple inheritance to co-exist gracefully.

In this example, each of these brands has a method eval( ) which evaluates that node of the AST, as in the code in Fig. 3.5. Suppose we intend DebugNode to act as a generic wrapper brand for each of the sub-brands of ASTNode. This can be implemented by using a dynamically-dispatched super call of the form ASTNode.super.eval( ) after performing the debug-specific functionality (in this case, printing the node's string representation). The prefix ASTNode.super means "find the first super-brand of this that implements ASTNode." At runtime, when eval( ) is called on an instance of DebugPlus, the chain of calls proceeds as follows: DebugPlus.eval( ) ⟼ DebugNode.eval( ) ⟼ Plus.eval( ). If the dynamically-dispatched super call behaved as an ordinary super call, it would fail—DebugNode has no super-brand.

Each of the DebugNode sub-brands implements its own eval( ) method that calls DebugNode.eval( ) with an ordinary super call. (This could be omitted if the language linearized method overriding based on the order of inheritance declarations, as described below.) Dynamic super calls are a generalization of ordinary super calls, when the qualifier brand is a required brand.

3.5.1 Multiple Inheritance and Method Names

The AST example ignored the details of simple and qualified method names, but we will now examine the effect of multiple inheritance on naming.

As mentioned in Sect. 3.4.1, method override is based on subclassing, rather than subtyping. Consequently, in the above example, DebugNode.eval is not in the same method family as ASTNode.eval and it would therefore have a different qualified name. Subclasses of DebugNode,

```
brand DebugNode requires ASTNode (
    method eval: ( ) ⇒ ASTNode =
        print(this.toString( ) ) ;
        ASTNode.super.eval  // dynamic super call
)

brand DebugPlus extends DebugNode, Plus (
    method eval: ( ) ⇒ ASTNode =
        DebugNode.super.eval  // ordinary super call
)
```

Figure 3.5: Implementing a mixin-like debug brand using dynamically-dispatched super calls.

such as DebugPlus, would inherit both ASTNode_eval and DebugNode_eval and would override the former method to call the latter.

It would be interesting to extend Unity to allow DebugNode.eval to be in the same method family as ASTNode.eval, particularly if the language also added some form of linearization semantics. For example, if DebugPlus inherited from DebugNode, ASTNode, this could mean that the DebugNode.eval definition is to take precedence. As this is largely an orthogonal issue, however, I have omitted linearization from the language.

3.5.2 Utility of requires

It may seem that the main benefit of requires is that it provides subtyping without sub-branding, which has already been implemented in many languages (e.g. [Black et al. 1986; Hutchinson 1987; Raj et al. 1991; Cook et al. 1990; Szyperski et al. 1993; Liskov et al. 1994; Chambers and the Cecil Group 2004; Johnsen et al. 2006]). In fact, Unity as described in the previous chapter already separated the notion of subtype and sub-brand! So, if this were the only benefit of requires, it would seem that it could be omitted from the language. But, as it turns out, the utility of requires lies in the fact that it establishes a *stronger relationship* than mere subtyping—it enforces the requirement that subclasses use a particular inheritance path in order to implement the required functionality. This in turn affects two seemingly orthogonal issues: dynamically-dispatched super calls and brand-private state.

Dynamically-dispatched super calls. In the AST node example, we saw that it was possible to implement mixin-like functionality using the dynamically-dispatched super call, where the call was dispatched to the brand that eventually implemented the required functionality. In a system without requires, however, designing a similar feature would be decidedly non-trivial (in fact, it is possible that method calls could easily become ambiguous). Thus, I argue that a) this is a useful feature (as illustrated in the example above) and b) a coherent language design would need to capture some notion of inheritance dependency in order to implement such a feature.

Brand-private state. In any language that separates sub-branding/subclassing and subtyping (using either nominal or structural subtyping), an interface cannot contain private members. Otherwise, superclasses would be able to access private members defined in subclasses—a violation of information hiding.

Unfortunately, this restriction can be problematic for defining binary methods such as the equals method; its argument type must contain those private members for the method be able to access them. But, for this type to contain private members, it must be tied to a particular class implementation, as only subclasses (as opposed to subtypes) should conform to this type.

Concretely, consider the following program (in a fictional Java-like syntax):

```
class A {
   private int i;
   boolean equals(A other) {
      ...    // can access other.i?
   }
}
class B subtypes A {
   ...    // declare i?
}
```

Suppose that the subtypes keyword provides nominal subtyping without inheritance (but without the additional constraints of requires). The question then arises: are private members considered when checking subtyping? If so, then B must declare a private field i. Unfortunately, this also means that A.equals can access B.i, which violates information hiding; one class should not be able to access private members defined in another class. On the other hand, if we assume that subtyping does not include private members, then A.equals cannot access other.i, which is problematic if the definition of equality depends on this field. An analogous problem occurs if structural subtyping is used.[12]

The problem can be avoided if inheritance or requires is used for types that contain binary methods. Since requires is tied to a particular class, if we change the above code to B requires A (or B extends A), then A.equals(A other) may safely access other.i, even if an object of type B is passed to this method. Note that an information hiding problem does not arise here—the private state has not been redefined in B, but is rather (eventually) inherited from A in the concrete B implementation that was passed in.

In summary, requires provides at least two benefits in addition to subtyping without sub-branding: it makes it possible to define a straightforward semantics for a form of dynamically-dispatched super call, and it makes possible the definition of brand-private state. The former is useful for defining mixin-like classes, and the latter is important for defining binary methods, such as equals.

[12] I am assuming here that brand-private state is the desirable semantics. If an object-private semantics were used, no problems would arise, but it would also be impossible to define the appropriate equals method.

3.5.3 Comparison to Other Languages

In this section, the AST example is encoded in other languages (those with single inheritance, mixins, or traits) and the resulting designs are compared to that in the example.

Single Inheritance

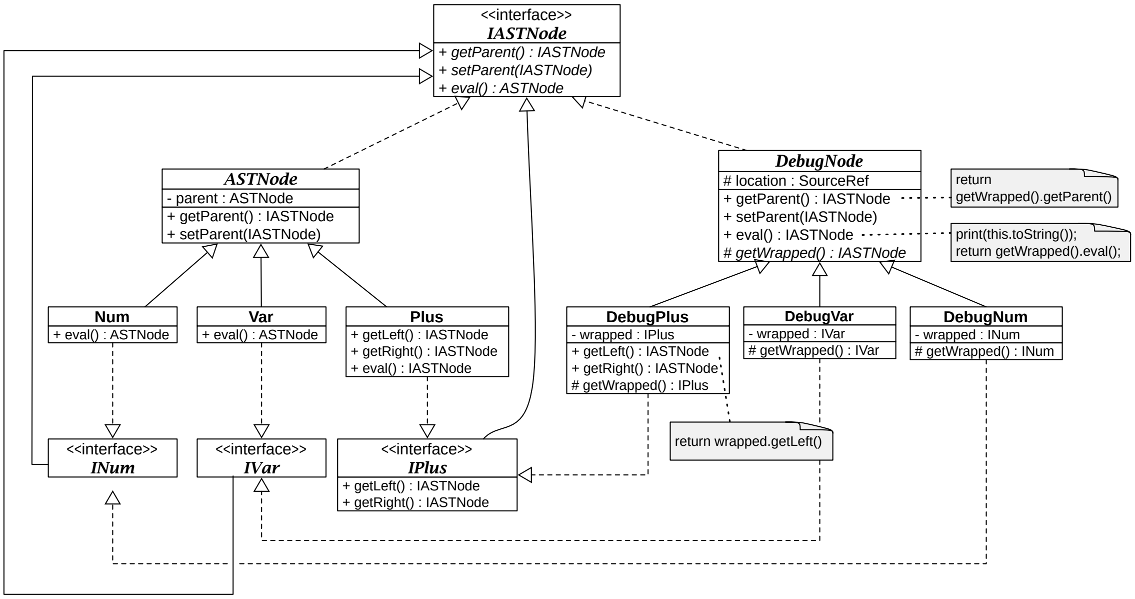

Figure 3.6: The example of Fig. 3.4 expressed in a Java-like language, resulting in a proliferation of interfaces and boilerplate code. The visibility modifiers '+', '-' and '#' indicate public, private and protected, respectively. Dashed lines represent `extends`; solid lines represent `implements`.

This example would be more difficult to express in a language with single inheritance. One straightforward design in a Java-like language is presented in Fig. 3.6. Multiple inheritance is simulated using interfaces for subtyping, and composition for dispatch. For instance, calls to `DebugPlus.getLeft()` are delegated to the wrapped `IPlus` object. The template method design pattern is used by `DebugNode` to implement `eval()` (subclasses override `getWrapped()`).

Note the addition of 4 new interfaces and the boilerplate code needed to implement getters, setters and delegation. The problem would be even worse if another dimension of behavior were to be added. Furthermore, the design has the problem that the getters and setters have to be public, since they are defined in an interface. For instance, the "parent" field in `ASTNode` is effectively fully visible, adversely affecting information hiding. Additionally, one would have to implement the visitor design pattern (not shown) to allow external traversal of the AST.

```
brand EnhancedDebugVar extends DebugVar (
    defLocation : SourceRef;
    method varLocToString: ( ) ⇒ string = . . . // create user−readable string from defLocation

    method eval: ( ) ⇒ ASTNode =
        print(this.varLocToString)
        DebugNode.super.eval
)
```

Figure 3.7: Adding a new sub-brand of DebugVar

Mixins

This example would be also difficult to express using mixins. Aside from the limitation that a total ordering must be specified during mixin composition [Ducasse et al. 2006], other issues arise. Suppose that in a variation of the previous example, we were to add a new sub-brand of DebugVar, EnhancedDebugVar. The intended semantics is that DebugVar.eval( ) prints the variable name, while EnhancedDebugVar.eval( ) prints the variable name and the location where the variable was defined in the source program. To implement this, we put the functionality of storing and printing a variable location into EnhancedDebugVar. Concretely, we would add the code in Fig. 3.7.

A possible translation of this example into Jam (an extension of Java with mixins [Ancona et al. 2003]) is shown in Fig. 3.8. First, we must create mixin equivalents of DebugNode and DebugVar, which will be prefixed with M. Since mixins cannot inherit from one another, MDebugVar would not be able to express an explicit relationship with MDebugNode (nor Var), but would instead have to declare location and eval( ) (and left, right, etc.) as required members.

This, in turn, leads to two problems. First, MDebugVar cannot be treated as a subtype of MDebugNode, which is a serious loss of expressiveness as compared to Unity. Second—and more significantly—supposing that external methods were to be integrated with mixins, it would be impossible to write an external method for MDebugNode and override it for MDebugVar, because there is no relationship between the two mixins. That is, we may wish to write methods MDebugNode.*m* and MDebugVar.*m* (its override), and have the mixin MEnhancedDebugVar inherit this latter definition. Instead, the definition of *m* must be pushed down to MEnhancedDebugVar, which creates problems for code reuse. In particular, suppose that method *m* and MEnhancedDebugVar are independent extensions that have no knowledge of each other. In a mixin world, external method definitions cannot be truly modular extensions.

The heart of the problem is that mixins are defined in isolation—though they can be composed, they cannot be subclasses (or even subtypes) of one another. Our proposed solution does bear some similarity to mixins, but additionally provides subtyping, design intent (through requires) and (no-diamond) multiple inheritance.

```
mixin MDebugNode {
   inherited ASTNode parent;
   SourceRef location;
   inherited ASTNode eval() {
      println(this.toString());
      return super.eval();
   }
}
mixin MDebugVar {
   inherited SourceRef location;
   inherited ASTNode left, right;

   inherited String toString() { ... } // use location, left and right fields
}
mixin MDebugVarEnhancer {
   inherited ASTNode left, right;
   defLocation : SourceRef;

   inherited ASTNode eval() {
      ... // call super.eval and use defLocation to print out debug info
   }
}

// psuedo–syntax; in Jam, would have to create intermediary classes
class EnhancedDebugVar =
   MDebugVarEnhancer extends (MDebugVar extends (MDebugNode extends Var)) { }
```

Figure 3.8: Rewriting parts of the AST example using Jam-style mixins [Ancona et al. 2003].

Traits

Traits could be used to express this example, but their lack of state results in an information-hiding problem with accessors, a problem similar to that of the single inheritance design. The stateful traits design [Bergel et al. 2008] does not provide a mechanism for true information hiding, as state can always be “unhidden” within classes composing the trait. In that design, all state is effectively “protected” (in the C++ sense of the term).

A possible encoding of the AST example into a language with traits is shown in Fig. 3.9. Note the duplication of accessor methods, and the fact that that traits such as T_ASTNode, may *not* define “state” that is private to the trait—by definition, the composing class must implement the associated accessor methods.

Scala traits

Scala traits are fusion of ordinary traits and mixins. Unlike ordinary traits, Scala traits may contain state (thereby avoiding the information hiding problem), and unlike ordinary mixins,

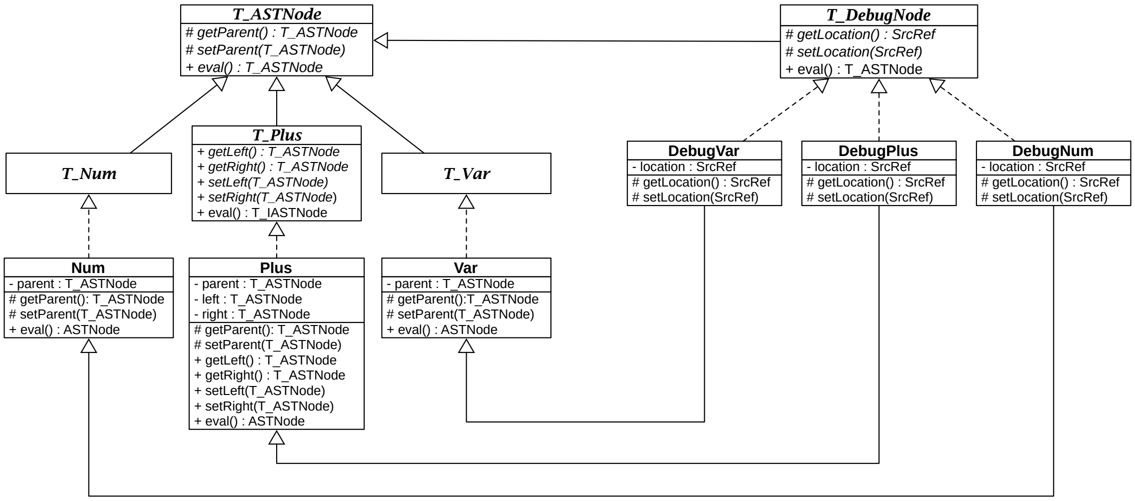

Figure 3.9: The example of Fig. 3.4 expressed in a language with traits (in the sense of [Schärli et al. 2003]). The visibility modifiers '+', '-' and '#' indicate public, private and protected, respectively.

Scala traits may define inheritance relationships. Accordingly, this particular example could be expressed quite elegantly in Scala. Unfortunately, the problems of diamond inheritance would again arise. In particular, if Scala supported any form of multimethod or external method, then a solution similar to that of Fortress or JPred would have to be employed for ensuring that multimethods were unambiguious. Concretely, the definition of a method similar to the defCheck external method would either be potentially ambiguious or unduly difficult to implement. And, as previously mentioned, to avoid the object initialization problem, Scala traits cannot have constructor parameters—a serious limitation.

Note that Scala also has a feature similar to what I have called a dynamically-dispatched super call. A method can be marked abstract override, which means that it is an override of a yet-to-be-inherited method. A super call within such a method has the same dispatch semantics as the Unity dynamic super call.

3.5.4 Discussion

Analogy to case analysis

The examples also illustrate that sub-branding, in addition to providing inheritance, defines semantic alternatives that may not overlap (such as Num, Var and Plus in the example above). Because they do not overlap, we can safely perform an unambiguous case analysis on them—that is, external dispatch. In other words, external dispatch in Unity is analogous to case-analyzing datatypes in functional programming.

Here, I discuss some additional design issues, such as encapsulation, comparison to traits, and potential Unity extensions.

Encapsulation and the Diamond Problem

As noted by Snyder, there are two possible ways to view inheritance: as an internal design decision chosen for convenience, or as a public declaration that a subclass is specializing its superclass, thereby adhering to its semantics [Snyder 1986].

Though Snyder believes that it can be useful to use inheritance without it being part of the external interface of a class, I argue that the second definition of inheritance is more appropriate. In fact, if inheritance is being used merely out of convenience (e.g., Vector extending Stack in the Java standard library), then it is very likely that *composition* is a more appropriate design [Bloch 2001]. For similar reasons, I do not believe a language should allow inheritance without subtyping—e.g., C++ private inheritance—as this can always be implemented using a helper class/brand whose visibility is restricted using the language's module system.

Nevertheless, if one takes the view that inheritance choices should *not* be visible to sub-brands (or subclasses), a form of the diamond problem can arise in Unity. In particular, suppose brand D extends B and C, C extends A, and B extends Object—a valid hierarchy (recall that condition *B1* makes a special exception for diamonds involving Object). Now suppose that B is changed to extend A, and the maintainer of B is unaware that brand D exists. Now A, B and C typecheck, but D does not. Thus, the use of inheritance can invalidate sub-brands, which violates Snyder's view of encapsulation.

This situation highlights the fact that, in general, requires should be favored over extends if a brand is intended to be reused.

Extensions

It would be possible to combine the proposed solution with existing techniques for dealing with the object initialization and modular multiple dispatch problems. A programmer could specify that a brand C, whose constructor takes no arguments, may be the root of a diamond hierarchy. Then, one would use the Scala solution for ensuring that C's constructor is called only once. To solve the multiple dispatch problem, if C is the owner of a method family m, the typechecker would ensure that m contained disambiguating definitions for the case of a diamond—the JPred and Fortress solutions.

One could also generalize dynamically-dispatched super calls so that they are *chained*, as in Scala [Odersky and Zenger 2005]. In Scala, a super call in a trait is dispatched to the next type in the linearization. In this way, traits can call sibling methods, which is a very powerful composition mechanism.

Finally, the language could include syntactic sugar to ease the definition of concrete brands. If C requires B, and both C and B have no-argument constructors, the compiler could automatically generate a brand *C*$concrete that extends both C and B; programmers could then more easily define external methods that dispatch on *C*$concrete.

Multiple Inheritance vs. Traits

As a motivation for the design of traits, Schärli et al. [Schärli et al. 2003] identified several problems with multiple inheritance; I describe here how our proposed solution addresses these issues. The problems are: 1) *conflicting features*: methods or variables are inherited along two different paths, particularly in cases of diamond inheritance; 2) *accessing overriding features*: using a single keyword (such as super) is insufficient to unambiguously identify inherited methods, so one must explicitly specify the superclass (or the language must linearize the class hierarchy); and 3) *factoring out generic wrappers*: programmers cannot use multiple inheritance to write reusable classes that wrap methods that will be implemented by future classes.

The first of these problems, conflicting features, is solved by *B1* (no diamonds) and the property of unique method names (described in Sect. 2.4.3). Problem (2) is a less important concern in a statically typed language, as programmers are already accustomed to specifying brand names for types. (This does indeed make it slightly more difficult to move methods to other brands, but refactoring tools make this task trivial.) Problem (3) is solved through dynamically-dispatched super calls, outlined above and demonstrated in Sect. 3.5.

3.6 Real-World Examples

In this section, real-world examples (in both C++ and Java) are presented that suggest that multiple inheritance, and diamond inheritance in particular, can be useful for code reuse. I also describe how these examples can be expressed in Unity.

3.6.1 C++ Examples

I examined several open-source C++ applications in a variety of domains and found many instances of virtual inheritance and inheritance diamonds. Here inheritance diamonds in two applications are described: Audacity[13] and Guikachu.[14]

Audacity

Audacity is a cross-platform application for recording and editing sounds. One of its main storage abstractions is the class BlockedSequence (not shown), which represents an array of audio samples, supporting operations such as cut and paste. A BlockedSequence is composed of smaller chunks; these are objects of type SeqBlock, depicted in Fig. 3.10 (a). One subclass of SeqBlock is SeqDataFileBlock, which stores the block data on disk. One superclass of SeqDataFileBlock is ManagedFile, an abstraction for temporary files that are de-allocated based on a reference-counting scheme. Since both ManagedFile and SeqBlock inherit from Storable (to support serialization), this forms a diamond with Storable at the top.

This particular diamond can be easily re-written in Unity (Fig. 3.10 (b)), since the sides of the diamond (SeqBlock and ManagedFile) are already abstract classes. (Compare to the example in Fig. 3.2, where new concrete brands had to be defined for the sides of the diamond.) Here, we would simply change the top two virtual inheritance edges to requires edges, and make SeqDataFileBlock inherit from Storable directly. This may even be a preferable abstraction; while

[13] http://audacity.sourceforge.net/

[14] http://cactus.rulez.org/projects/guikachu/

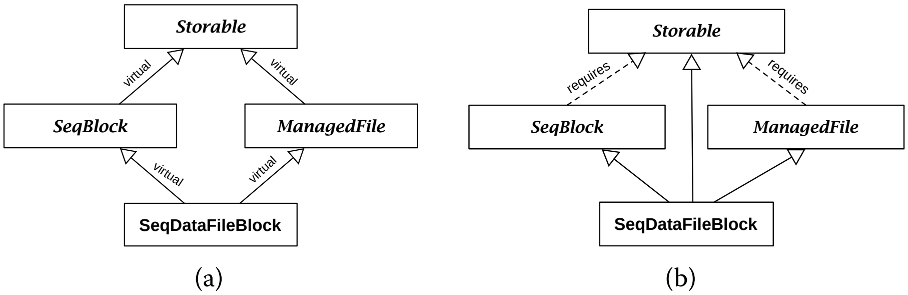

Figure 3.10: An inheritance diamond (a) in the Audacity application, and (b) the re-written class hierarchy in Unity. Abstract classes are set in italic.

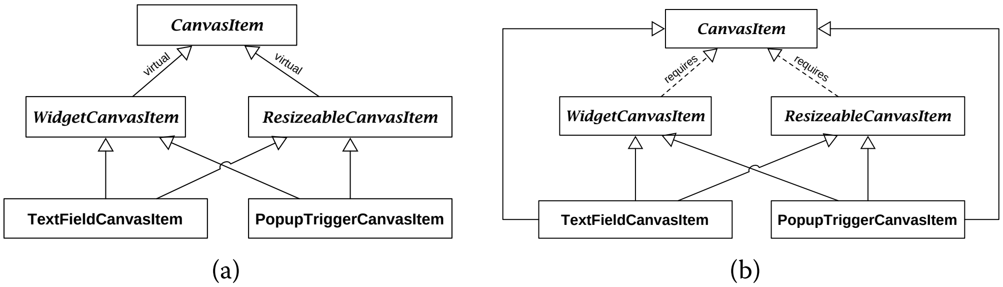

Figure 3.11: Two inheritance diamonds (a) in the Guikachu application and the classes re-written in Unity (b). Abstract classes and brands are set in italic.

in the original hierarchy SeqDataFileBlock is serializable by virtue of the fact that SeqBlock is serializable, in the new hierarchy we are making this relationship explicit.

Guikachu

Guikachu is a graphical resource editor for the GNU PalmOS SDK. It allows programmers to graphically manipulate GUI elements for a Palm application in the GNOME desktop environment. In this application, I found 10 examples of diamonds that included the classes CanvasItem, WidgetCanvasItem, and ResizeableCanvasItem. CanvasItem is an abstract base class that represents items that can be placed onto a canvas, while objects of type WidgetCanvasItem and ResizeableCanvasItem are a type of widget or are resizeable, respectively.

Figure 3.11(a) shows two of these 10 diamonds, formed by TextFieldCanvasItem and PopupTriggerCanvasItem, respectively. The hierarchy was likely designed this way because there exist GUI elements that have only one of the two properties. For instance, GraffitiCanvasItem and LabelCanvasItem (not shown) are not resizeable, but they are widgets. In contrast, the class FormCanvasItem (not shown) is resizeable, but is not a widget.

In this application, I also observed the use of the C++ virtual inheritance initializer invocation mechanism: TextFieldCanvasItem (for instance) directly calls the initializer of CanvasItem, its grandparent. As previously described, when initializing TextFieldCanvasItem, the initializer calls from WidgetCanvasItem and ResizeableCanvasItem to CanvasItem are ignored. In this application, the initializers happen to all perform the same operation, but this invocation semantics could introduce subtle bugs as the application evolves.

The corresponding Unity brand hierarchy is displayed in Fig. 3.11 (b); note its similarity to that of Fig. 3.10 (b). Essentially, the virtual inheritance is replaced with requires and each of the brands at the bottom of the diamond inherit from all three of WidgetCanvasItem, ResizeableCanvasItem, *and* CanvasItem. The Unity design has the advantage that constructor calls do not occur more than one level up the hierarchy, and no constructor calls are ignored.

This example illustrates how a program could be translated from C++-style multiple inheritance to Unity-style. In particular, virtual inheritance would be replaced by requires, and new concrete brands would be defined as necessary (changing instantiations of the now-abstract brand to instantiations of the new concrete brand). Note that constructor calls can be easily generated for the new concrete brands, as C++ requires a call from the bottom of the diamond to the top of the diamond when virtual inheritance is used (such a constructor call would be necessary for the new concrete brand, as it would directly extend the brand at the top of the diamond).

3.6.2 Java Example: Eclipse JDT

The Eclipse JDT (Java Development Tools) provides an example of where multiple inheritance could be useful for Java programs. In the JDT, every AST node contains *structural properties*. A node's structural properties allow uniform access to its components. For example, DoStatement has two fields of type StructuralPropertyDescriptor: EXPRESSION_PROPERTY and BODY_PROPERTY. To get the expression property of a DoStatement object, the programmer may call ds.getExpression() or ds.getStructuralProperty(DoStatement.EXPRESSION_PROPERTY). Structural property descriptors are often used to specify how AST nodes change when a refactoring is performed.

Through inspection of the JDT code, I found that there was a great deal of duplication among the code for getting or setting a node property using the structural property descriptors. For example, 19 AST classes (for instance, AssertStatement and ForStatement) have getExpression/setExpression properties. As a result, in the method internalGetSetChildProperty (an abstract method of ASTNode), there are 19 duplications of the following code:

```
if (property == EXPRESSION_PROPERTY) {
   if (get) {
      return getExpression();
   } else {
      setExpression((Expression) child);
      return null;
   }
} else if (property == BODY_PROPERTY) {
   . . .   // code for body property
  }
}
```

Additionally, there are duplicate, identical definitions of the EXPRESSION_PROPERTY field. Without a form of multiple inheritance, however, it is difficult to refactor this code into a common location—DoStatement, for example, already has the superclass Statement. In Unity, on the other hand, the programmer could create an abstract helper brand ExprPropertyHelper that requires ASTNode. This new brand would contain the field definition and an override of internalGetSetChildProperty. DoStatement would then inherit from both Statement and ExprPropertyHelper and would have the following body for internalGetSetChildProperty:

```
if (property == BODY_PROPERTY)
   . . .    // code for body property
else
   ExprPropertyHelper.super.internalGetSetChildProperty(property, get, child)
```

Finally, this is a scenario where mulitple dispatch would be beneficial. The framework defines various visitors for traversing an AST; these could be omitted in favor of external methods or multimethods, which are more extensible.

Discussion

Overall, the real-world examples suggest that multiple inheritance can be useful, and that even diamond inheritance is used in practice. I have shown that the inheritance diamonds can be easily translated to Unity and that the resulting designs offer some benefits over the original ones.

3.7 Formal System

In this section, I describe the highlighted portions of Sect. 2.5.

The grammar additions needed to support multiple inheritance are relatively minor (Fig. 2.15, p. 37). Brand declarations additionally have a requires clause that is similar to the extends clause:

$$\textit{brand-decl} ::= \mathsf{brand}\ B(\tau; \overline{\textit{m-decl}})\ \mathsf{extends}\ C_1, \ldots, C_n\ \mathsf{requires}\ \overline{D}$$

External method blocks (*method-decl*) have an owner brand that is specified before the method name. Finally, there is a new expression form that has already been described in the examples:

$$e ::= \ldots \mid e.B.\mathsf{super}.q \ \ldots \mid$$

This is the syntactic form for the dynamically-dispatched super call, which was illustrated in the AST example.

3.7.1 Static Semantics

This section describes the multiple inheritance changes to the subtyping judgement, then those regarding the typechecking of brands, external methods, and expressions.

Subtyping. Only the subtype judgement (≤) has an additional rule:

$$\frac{\begin{array}{c} B(\tau;Q)\ \text{extends}\ \overline{B}\ \text{requires}\ C_1,\ldots,C_n \in \Sigma \\ \Gamma \vdash M_1 \trianglelefteq M_2 \end{array}}{\Gamma \vdash B(M_1) \le C_i(M_2)}\ (\textsc{Sub-Requires})$$

This rule provides property (2) of Def. 3.1; B branded objects are subtypes of C_i branded objects. The rule is safe, due to properties (1) and (3) of the aforementioned definition.

Brand declarations. Recall the rules for typechecking top-level declarations: Tp-Brand-Decl and Tp-Ext-Method (Fig. 2.18, p. 42). For typechecking brand declarations, the auxiliary judgement **inherit-ok** was used:

$$(\textsc{Tp-Inherit})$$
$$\frac{\begin{array}{c} ① \ \overline{C} \in \Sigma \qquad ② \ \overline{D} \in \Sigma \qquad ③ \ D_i \notin \overline{C} \ \ (\forall i \in 1..m) \\ ④ \ \forall i,j.\, i \neq j. \nexists D'.\, C_i \sqsubseteq_\Sigma D' \ \text{and}\ C_j \sqsubseteq_\Sigma D' \quad (D' \neq \text{Object}) \\ ⑤ \ C_i\ \text{requires}\ E \in \Sigma \ \ \text{implies}\ \ \exists k.\ C_k \sqsubseteq_\Sigma E \ \text{or}\ D_k \sqsubseteq_\Sigma E \ \ (\forall i \in 1..n) \\ ⑥ \ D_i\ \text{requires}\ E' \in \Sigma \ \ \text{implies}\ \ \exists k.\ C_k \sqsubseteq_\Sigma E' \ \text{or}\ D_k \sqsubseteq_\Sigma E' \ \ (\forall i \in 1..m) \\ ⑦ \ \forall i,j.\, \forall q.\, \mathit{mtype}_\Sigma(q, C_i) = \rho \ \text{and}\ \mathit{mtype}_\Sigma(q, C_j) = \rho' \ \text{implies}\ i = j \end{array}}{\vdash_\Sigma \ B\ \text{extends}\ C_1,\ldots,C_n\ \text{requires}\ D_1,\ldots,D_m \in \Sigma \ \ \textbf{inherit-ok}}$$

That judgement ensures that the declared superclasses exist in Σ, that there are no inheritance diamonds (premise 4), that requires is propagated down the inheritance hierarchy (premises 5 and 6), and that there are no methods with the same name in two inherited classes.[15]

For typechecking the bodies of internal methods (*decl* **body-ok**), the requires clause is taken into account (Fig. 2.20, p. 44):

$$(\textsc{Brand-Decl-Body})$$
$$\frac{\text{fieldType}_\Sigma(\overline{D}) = \overline{\sigma}' \qquad \text{this} : B(M_i), \text{fields} : \sigma \wedge \overline{\sigma}' \vdash_\Sigma \ e_i : \tau_i}{\text{brand}\ B(\sigma; q_i \ B(M_i) : \tau_i = e_i \ ^{i \in 1..n})\ \text{extends}\ \overline{C}\ \text{requires}\ \overline{D}\ \textbf{body-ok}}$$

The second premise looks up the field type of B's required brands, naming them (the list) $\overline{\sigma}'$. Then, when typechecking the method body (the third premise), we can safely assume that the fields of the receiver (the special variable fields) actually contain these required-brand fields $\overline{\sigma}'$. This is sound for the same reason that the subtyping rule Sub-Requires is sound: property (1) of Def. 3.1 ensures that objects cannot be instantiated from a brand that has a requires clause. (This property is enforced when typechecking the object-instantiation expression, described in a subsection below.)

[15] Note that this property will always hold if an elaboration of the form outlined in Sect. 2.4.3 is used.

External method blocks. The rule Tp-Ext-Method (Fig. 2.18, p. 42) typechecks an external method family:

$$\textsc{(Tp-Ext-Method)}\quad \frac{\begin{array}{c}① \; B \neq \mathsf{Object} \qquad ② \; \overline{C} \text{ distinct} \qquad ③ \; C_i \sqsubseteq_\Sigma B \;\; (\forall i \in 1..n) \\ ④ \; \nvdash_\Sigma \overline{C}.q \textbf{ internal} \qquad ⑤ \; \Sigma' = \Sigma, \mathsf{method}\; B.q(\overline{C}.q : \overline{\rho}) \qquad ⑥ \; \vdash_{\Sigma'} \overline{C}.q : \overline{\rho} \textbf{ override-ok}\end{array}}{\Sigma \vdash \mathsf{method}\; B.q(\overline{C}.q : \overline{\rho}) \textbf{ ok}}$$

Here, premise (1) enforces the first part of condition *E3*: the owner brand cannot be Object. The second part of condition *E3* (each method declaration is defined on a sub-brand of the owner brand) is enforced by premise (3).

Expressions. The changes to the rules for typechecking expressions are fairly minor. The first premise of Tp-New-Obj (Fig. 2.21, p. 45) enforces property (1) of Def. 3.1—brands with a requires clause may not be instantiated.

The rule Tp-With adds new simple to qualified mappings after an object has been created. To simplify the formal system, the *mtype* judgement does not consider required brands, only extended brands. This does not cause any problems for expressiveness, however, as a brand can always be coerced to its required brand via the subsumption rule.

But, when setting up the mapping, we may wish to add methods from both inherited and required brands. Thus, the rule permits any mapping for which there exists *mtype* for the brand itself or one of its required brands:

$$\textsc{(Tp-With)}\quad \frac{\begin{array}{c}\Gamma \vdash e : B(M) \qquad B \;\mathsf{requires}\; \overline{D} \in \Sigma \qquad \overline{n} \notin M \qquad \overline{n} \text{ distinct} \\ \exists C \in \{B, \overline{D}\}.\; \mathit{mtype}_\Sigma(q_i, C) : \rho_i \;\; (\forall i \in 1..n)\end{array}}{\Gamma \vdash e \;\mathsf{with}\; n_i \hookrightarrow q_i{}^{i\in 1..n} : B(M, n_i : \rho_i{}^{i \in 1..n})}$$

The final multiple inheritance addition is rule Tp-Invoke-Super (also in Fig. 2.21) which typechecks the dynamically-dispatched super call. Here, we check that the brand qualifier *C* is require'd by the object's brand and perform an *mtype* lookup using *C*. Note the similarity of Tp-Invoke-Super to Tp-Invoke-Nom, which typechecks (ordinary) qualified method invocation.

3.7.2 Dynamic Semantics

There are fewer changes to the dynamic semantics: there is a congruence and computation rule for dynamically-dispatched super calls. (Fig. 2.25, p. 48). The congruence rule is standard; the computation rule, E-Super-Invk-Val, is:

$$\frac{\exists \text{ unique } C'.\; \mathit{super}_\Delta(B \text{ as } C) = C' \qquad \mathit{lookup}_\Delta(q, C') = e}{\widehat{B}(v; \overline{n \hookrightarrow q}).C.\mathsf{super}.q \longmapsto_\Delta \{\widehat{B}(v; \overline{n \hookrightarrow q})/\mathsf{this}, v/\mathsf{fields}\}\; e}$$

Here, we use (the unique result of) the *super* auxiliary judgement to find the appropriate class C'; the method *q* is then looked up within C'. This auxiliary judgement is:

$$\frac{B \text{ extends } D \in \Delta \qquad D \sqsubseteq_{\Delta} C}{\mathit{super}_{\Delta}(B \text{ as } C) = D} \qquad \frac{\begin{array}{c}\nexists D' \sqsubseteq_{\Delta} C.\, B \text{ extends } D' \in \Delta \qquad B \text{ extends } \overline{E} \in \Delta \\ \exists k.\, \mathit{super}_{\Delta}(E_k \text{ as } C) = D\end{array}}{\mathit{super}_{\Delta}(B \text{ as } C) = D}$$

That is, *super* finds the first super-brand of B that is also a subtype of C (i.e., the first parent that fulfills the requires C enforced by Tp-Invoke-Super). The judgement *super* uses the sub-brand judgement on runtime contexts, $\sqsubseteq_{\Delta}$, this latter judgement mirroring $\sqsubseteq_{\Sigma}$ with the exception that Δ is used rather than Σ:

Definition 3.2 ($\sqsubseteq_{\Delta}$ judgement).

The judgement $B \sqsubseteq_{\Delta} C$ is defined by inference rules identical to those of $B \sqsubseteq_{\Sigma} C$ (i.e., Sub-Brand-Refl, Sub-Brand-Trans, and Sub-Brand-Decl in Fig. 2.16), except that the runtime context Δ is used instead of Σ.

3.7.3 Modularity

As mentioned in Sect. 1.1, modular typechecking is an essential property for a practical language. In its absence, scalability and extensibility issues will immediately arise, particularly when multiple developers are working on a project. With a modular type system, programmers can be assured that their modules will typecheck when incorporated into a larger program and clients of a module can be shielded from changes to a module's internals, when those changes do not affect its interface.

I define a typechecking algorithm as *modular* if each module in the program can be typechecked using only the interfaces of the other modules on which it *statically* depends. Moreover, if a module typechecks in isolation, this fact should not change when it is combined with other modules. From this it follows that the linker cannot perform any typechecking; it mainly performs a consistency check to ensure that a definition is present for all modules that have been assumed to exist in the program.

In the Unity calculus, each top-level declaration (i.e., brand or external method) is assumed to be in its own module. Recall that top-level declarations contain a context Σ which include all definitions on which the declaration statically depends. This context is of key importance for the modularity proof, as Σ is precisely the interfaces of the other modules for which the module in question assumes a definition exists.

From this it follows that typechecking is modular if each declaration is typechecked under its declared context Σ and if the declaration will always typecheck under any context Σ' that contains at least all the declarations in Σ.

The calculus has been carefully designed so that the proof of modularity is straightforward:

Theorem 3.1. Typechecking top-level elements declarations *decl* is modular; i.e., typechecking such elements only involves examining the signatures Σ on which *decl* statically depends.

Proof. Follows from the fact that top-level elements are typechecked under their declared context Σ_0. The only rules that examine the entire linearized program context Σ are Tp-Decl-Ok

$$\textsc{(Sigma-Wf-Base)}\quad \frac{}{\cdot\ \mathbf{ok}} \qquad\qquad \textsc{(Sigma-Wf-Decl)}\quad \frac{\mathit{decl\text{-}type}_{name} \notin \Sigma \qquad \Sigma\ \mathbf{ok} \qquad \Sigma \vdash \mathit{decl\text{-}type}\ \mathbf{ok}}{\Sigma, \mathit{decl\text{-}type}\ \mathbf{ok}}$$

Figure 3.12: Well-formed judgement for static context Σ

and Tp-Expr-Ok, in the premise $\Sigma \supseteq \Sigma_0$. This step is analogous to a linking phase in which imported declarations are resolved. Since checking set inclusion does not involve typechecking of any kind, this check adheres to the definition of modular typechecking. □

3.7.4 Type Safety

The full proof of type safety is provided in Appendix A; the main results are summarized here. First, we define the properties of well-formed context Σ (Fig. 3.12), which includes some of the same properties checked by Tp-Decl-Ok. Declarations must have unique names, and each declaration must be well-typed under the context of the types that precede it.

The type safety theorems assume a correspondence between the runtime context Δ and the brand and method context Σ. This ensures that the runtime context, which does not contain type information, is consistent with the static typing context, which does not contain any code. Formally, this correspondence is defined as follows:

Definition 3.3 (Context consistency relation).
The judgement $\Delta : \Sigma$ is defined by the following inference rules:

$$\textsc{(Delta-Wf-Empty)}\quad \frac{}{\cdot\ :\ \cdot} \qquad \textsc{(Delta-Wf-Brand)}\quad \frac{\begin{array}{c}\Sigma = \Sigma_0, \text{brand } B(\sigma; \{q_i : B(M_i) \Rightarrow \tau_i\ ^{i\in 1..n}\}) \text{ extends } \overline{C} \text{ requires } \overline{D}\\ \Delta_0 : \Sigma_0 \qquad \text{fieldType}_{\Sigma}(\overline{D}) = \overline{\sigma}\\ \text{this} : B(M_i), \text{fields} : \sigma \wedge \overline{\sigma} \vdash_{\Sigma} e_i : \tau_i \ \ (\forall i \in 1..n)\end{array}}{\Delta_0, B(q_i = e_i\ ^{i\in 1..n}) \text{ extends } \overline{C}\ :\ \Sigma}$$

$$\textsc{(Delta-Wf-Method)}\quad \frac{\begin{array}{c}\Sigma = \Sigma_0, \text{method } q(q : B_i(M_i) \Rightarrow \tau_i\ ^{i\in 1..n}) \qquad \Delta_0 : \Sigma_0\\ \text{this} : B_i(M_i) \vdash_{\Sigma} e_i : \tau_i \ \ (\forall i \in 1..n)\end{array}}{\Delta_0, \text{method } q(B_i.q = e_i\ ^{i\in 1..n})\ :\ \Sigma}$$

Type safety is proved using the standard progress and preservation theorems. These theorems each depend on weakening properties of the various judgements under a larger context Σ. For example, we have the following:

Lemma 3.1 (Weakening for sub-brand judgement).
If Σ_0 **ok** and $B \sqsubseteq_{\Sigma_0} C$ and Σ **ok** and $\Sigma \supseteq \Sigma_0$ then $B \sqsubseteq_{\Sigma} C$.

Proof. See proof of Lemma A.3 (p. 130). □

A similar weakening property holds for the subtyping and typing relations. Additionally, the *decl-type* **ok** judgement can be weakened so that we have the following:

Lemma 3.2 (Declarations are well-typed under their containing context).
If Σ **ok** and *decl-type* $\in \Sigma$ then $\Sigma \vdash$ *decl-type* **ok**.

Proof. See proof of Lemma A.14 (p. 135). □

The proof of the above lemma makes use of condition *E1*—since external method definitions q must all appear in the same block that the family q is introduced, extensions of the context Σ cannot contain new (external) definitions for q.

An auxiliary lemma is used for the weakening lemmas, which allows us to conclude that two brands B and C must have a common ancestor that is a strict subtype of Object if an *mtype* derivation exists for both B and C for the same method q:

Lemma 3.3 (Common method implies common ancestor).
If Σ **ok** and $mtype_{\Sigma}(q, B)$ and $mtype_{\Sigma}(q, C)$ then there exists some $D \neq \text{Object}$ such that $B \sqsubseteq_{\Sigma} D$ and $C \sqsubseteq_{\Sigma} D$.

Proof. See proof of Lemma A.2 (p. 130). □

Conditions *E1*–*E3* are all used in the proof of this lemma.

Progress

For progress, we prove a lemma that states that if method q has type ρ in B (either declared or inherited), then a runtime context Δ consistent with the static context Σ contains a unique method body for q:

Lemma 3.4 (*lookup* defined on well-typed objects).
If Σ **ok** and $mtype_{\Sigma}(q, B) = \rho$ and $\Delta : \Sigma$, then $lookup_{\Delta}(q, B) = e$, for some unique e.

Proof. By induction on the derivation $mtype_{\Sigma}(q, B)$. See proof of Lemma A.31 (p. 139). □

This lemma has several interesting cases. For internal and external methods, we use the fact that an internal and external method cannot both exist for a particular brand—condition *E2*. For the inductive step, we use the fact that a class cannot inherit a method with the same name from two distinct classes—premise (7) of Tp-Inherit. For the inductive case MType-Inh, we use condition *E3* to show that at most one external method definition is inherited from superclasses (i.e., the same external method cannot be defined for two different superclasses of a class C). Finally, because of the no-diamond property (condition *B1*), the same internal method cannot be inherited from two distinct superclasses.

Now we are ready to prove the standard progess theorem. Note that this theorem has a corresponding lemma for expressions, Lemma A.35 (p. 141), where we prove the more interesting cases. The most interesting case is method invocation, which simply uses the previous lemma.

Theorem 3.2 (Progress [programs]).
If Σ **ok** and $\Sigma \vdash p$ **ok**, then one of the following cases holds:

1. p is a value; or
2. for any Δ such that $\Delta : \Sigma$, there exist p' and Δ' such that $p \mid \Delta \longmapsto p' \mid \Delta'$.

Proof. See proof of Theorem A.1 (Sect. A.4, p. 143). □

Preservation

For proving preservation, we first prove standard inversion lemmas for subtyping and typing (Lemmas A.28 and A.29). Next, we prove a lemma that states that the result of *mtype* is unique:

Lemma 3.5 (*mtype* is a function).
If Σ **ok** and $mtype_\Sigma(q, B) = \rho_1$ and $mtype_\Sigma(q, B) = \rho_2$, then $\rho_1 = \rho_2$.

Proof. By simultaneous induction on the two *mtype* derivations. See proof of Lemma A.39 (p. 143). □

The lemma is proved using conditions *E1*–*E3*, as well as the no-diamond property (condition *B1*).

The next key preservation lemma proves that the result of the *lookup* judgement is consistent with the result of *mtype*:

Lemma 3.6 (Result of *lookup* is well-typed).
If Σ **ok** and $\Delta : \Sigma$ and $mtype_\Sigma(q, C) = N \Rightarrow \tau$ and $lookup_\Delta(q, C) = e_0$, then
$\mathsf{this} : \sigma_c, \mathsf{fields} : \sigma_f \vdash_\Sigma e_0 : \tau$.
for some σ_c and σ_f such that $C(N) \leq \sigma_c$ and $\text{fieldWithReq}_\Sigma(C) \leq \sigma_f$.

Proof. By induction on $lookup_\Delta(q, C)$. See proof of Lemma A.40 (p. 144). □

The base cases are straightforward, but the inductive case (Lookup-Inh) makes use of the *mtype* uniqueness lemma above (Lemma 3.5), to map a *lookup* derviation to a corresponding *mtype* derivation.

Now we are ready to state the main preservation theorem:

Theorem 3.3 (Preservation [programs]).
If Σ **ok** and $\Sigma \vdash p$ **ok** and $\Delta : \Sigma$ and $p \mid \Delta \longmapsto p' \mid \Delta'$, then there exists a Σ' such that (a) Σ' **ok**; and (b) $\Delta' : \Sigma'$; and (c) $\Sigma' \vdash p'$ **ok**.

Proof. See proof of Theorem A.2 (Sect. A.5, p. 149). □

As with progress, preservation also has a lemma for of expressions (Lemma A.42, p. 145), where most of the interesting reasoning lies, particularly the cases of method invocation.

3.8 Related Work

Here, I revisit the JPred and Fortress solutions and describe other closely related work: mixins, traits, and stateful traits. Comparison to additional, less closely related work appears in Sect. 5.4.

JPred and Fortress

As mentioned in Sect. 3.3, JPred [Frost and Millstein 2006] and Fortress [Allen et al. 2007] perform modular multimethod typechecking by requiring that programmers provide disambiguating methods, some of which may never be called. However, the JPred and Fortress dispatch semantics may be more expressive than that of Unity. In Unity, recall that in the class hierarchy Fig. 3.2, the abstract class InputStream may not override a Stream method externally, because it is not a subclass of Stream. In contrast, if this hierarchy were expressed in JPred or Fortress, a multimethod defined on Stream could be overridden by either InputStream or OutputStream. Note, however, that programmers can achieve a similar effect in Unity by having concrete classes call helper methods (which can be defined externally) in the abstract classes.

Mixins

Mixins, also known as abstract subclasses, provide many of the reuse benefits of multiple inheritance while fitting into a single inheritance framework [Bracha and Cook 1990; Ancona and Zucca 1996; Flatt et al. 1998; Findler and Flatt 1999; Ancona et al. 2003; Bettini et al. 2004]. While mixins allow defining state, they have two drawbacks: they must be explicitly linearized by the programmer and they cannot inherit from one another (though most systems allow expressing implementation dependencies, such as abstract members). If mixin inheritance were allowed, this would be essentially equivalent to Scala traits, which *do* have the object initialization problem. Additionally, the lack of inheritance has the consequence that mixins do not integrate well with multiple dispatch; multiple dispatch requires an explicit inheritance hierarchy on which to perform the dispatch.

Traits

Traits were proposed as a mechanism for finer-grained reuse, to solve the reuse problems caused by mixins and multiple inheritance [Ducasse et al. 2006; Fisher and Reppy 2004; Odersky and Zenger 2005]. In particular, the linearization imposed by mixins can necessitate the definition of numerous “glue” methods [Ducasse et al. 2006]. This design avoids many problems caused by multiple inheritance since fields may not be defined in traits.

Unfortunately, this restriction results in other problems. In particular, non-private accessors in a trait negatively impact information hiding: if a trait needs to use state, this is encoded using abstract accessor methods, which must then be implemented by the class composed using the trait. Consequently, it is impossible to define “state” that is private to a trait—by definition, all classes reusing the trait can access this state. (We will see an example of this in Sect. 3.5 below.) Additionally, introducing new accessors in a trait results in a ripple effect, as all client classes must now provide implementations for these methods [Bergel et al. 2008], even if there are no other changes.

In contrast, Unity allows a brand to multiply inherit other brands, which may contain state. In particular, a brand may extend other concrete brands, while in trait systems, only traits may be multiply inherited.

Stateful Traits

Stateful traits [Bergel et al. 2008] were designed to address the aforementioned problems with stateless traits. But, as previously mentioned, this language does not address the problem of a correct semantics for object initialization in the presence of diamonds. Additionally, stateful traits do not address the information hiding problem, as they have been designed for maximal code reuse. In this design, state is hidden by default, but clients can "unhide" it, and may have to resort to merging variables that are inherited from multiple traits. While this provides a great deal of flexibility for trait clients, this design does not allow traits to define private state.

3.9 Conclusions

In this chapter, I have shown how a small modification to the rules of traditional multiple inheritance can reap great rewards in terms of program reasoning. In particular, the problems of object initialization and modular typechecking of external methods disappear in a language that disallows diamond inheritance. I have also shown that the expressiveness of diamond inheritance can be recovered through use of the requires construct, even in real-world examples.

The type safety of the language was summarized, affirming hypothesis I. The modularity proof in Sect. 3.7.3 is the evidence for hypothesis V. The multiple inheritance design and the comparison to related systems (Java-style multiple interface inheritance, mixins and traits) illustrated the expressiveness of the Unity design (Sect. 3.5), a design that satisfies hypothesis VI. Finally, the C++ examples showed the feasibility of a systematic translation of diamond inheritance into Unity multiple inheritance, which supports hypothesis VII.

Chapter 4

Empirical Results

> "Give your evidence," said the King; "and don't be nervous, or I'll have you executed on the spot."
>
> Lewis Carroll (*Alice's Adventures in Wonderland*)

In this dissertation, I have proposed a combination of nominal and structural subtyping, in order to obtain the flexibility and expressiveness benefits of the latter typing discipline. To support the claim that structural subtyping is beneficial, I presented examples and applications to real-world situations (Sections 2.2 and 2.3).

Many in the research community agree with this view; as mentioned in Chapter 1, structural subtyping has been extensively studied in a formal setting (e.g., [Cardelli 1988; Bruce et al. 2003; Fisher and Reppy 1999; Leroy et al. 2004; Malayeri 2009a]). And yet, structural subtyping is not used in any mainstream object-oriented programming language—perhaps due in part to the lack of evidence of its utility. Accordingly, I considered the following question: what empirical evidence could show that structural subtyping can be beneficial?[1]

4.1 Empirical Criteria

Let us consider the characteristics that a nominally-typed program might exhibit that would indicate that it could benefit from structural subtyping. First, the program might systematically make use of a subset of methods of a type, with no nominal type corresponding to this method set. A particular such implicit type might be used repeatedly throughout the program. Structural subtyping would allow these types to be easily expressed, without requiring that the type hierarchy of the program change. This is particularly beneficial when a nominal hierarchy cannot be changed (due to lack of access to or control of the applicable source code), as changing a nominal hierarchy generally requires changes to the intended subtypes. For example, in Java, to express the fact that class C implements interface I, C's source must be modified.[2]

[1] The primary technical contributions of this chapter appeared in [Malayeri and Aldrich 2009b].

[2] This is no accident; fully-general retroactive subtyping leads to inherent modularity problems, particularly when nominal types can be used in runtime dispatch. See Sect. 5.2 for a discussion of these issues.

Second, there might be methods in two different classes that share the same name and perform the same operation, but that are not contained in a common nominal supertype. There are a number of reasons why such a situation might occur, such as oversight on the part of the original designers. This is particularly likely when the original code did not need to make use of the implicit interface induced by these common methods. Alternatively, perhaps such a need did exist, but programmers resorted to code duplication rather than refactoring the type hierarchy—possibly because the source code was not accessible or could not be changed. On the other hand, with structural subtyping, the two classes in question would automatically share a common supertype consisting of the shared methods.

Or, programs might use the Java reflection method Class.getMethod() to call a method with a particular signature in a generic manner. For instance, we may wish to write a method *m* that can be passed as an argument any object that contains a "String getName()" method. In nominally typed languages, this can generally be achieved only through dynamic means such as reflection; in contrast, structural subtyping provides such a capability in a statically-checkable manner.

Finally, suppose a programmer is faced with the challenge of writing a class *C* that only supports a subset of its declared interface *I*. But, such a super-interface does not exist and cannot be defined, perhaps due to library use. One possible implementation strategy is simply throw an exception (e.g., UnsupportedOperationException) when one of *C*'s unimplemented methods is called. In contrast, with structural subtyping, the intended structural super-interface could simply be used.

With all of these characteristics in mind, I performed several manual and automated analyses on (up to) 29 open-source Java programs. In the case of manual analyses, a subset of the subject programs was considered. Each of these analyses aimed to answer one question: are nominally-typed programs using implicit structural types? The result was that indeed they were; representing these types explicitly could therefore produce desirable characteristics, such as increased code reuse and decreased maintenance effort.

In the empirical evaluation, answers to the following questions were sought:

1. Does the body of a method use only a subset of the methods of its parameters? If so, structural types could ease the task of making the method more general. (Sect. 4.3)
2. If structural types are inferred for method parameters, do there exist inferred types that are used repeatedly, suggesting that they represent a meaningful abstraction? (Sect. 4.3.3)
3. How many methods always throw "unsupported operation" exceptions? In such cases, the enclosing classes support a structural supertype of the declared class type; the latter contains all of the declared and inherited methods of the class (regardless of their implementation, or lack thereof). (Sect. 4.4)
4. What is the nature and frequency of *common methods*? That is, sets of methods with identical names and signatures, but that are not contained in any common supertype of the enclosing classes. (Sect 4.5.1)
5. How many common methods represent an accidental name clash? (Sect 4.5.2)
6. Can structural subtyping reduce some types of code duplication? (Sect. 4.5.3)

7. Is there empirical evidence of a potential synergy between structural subtyping and external methods? (Sect. 4.6)
8. Do programs use reflection where structural types would be preferable? (Sect. 4.7)

Thus, a variety of facets of existing programs were considered. While none of these aspects is conclusive on its own, taken together, the answers to the above questions provide evidence that even programs written with a nominal subtyping discipline could benefit from structural subtyping. This study provides initial answers to the above questions; further study is needed to fully examine all aspects of some questions, particularly questions 5 and 6. Additionally, as mentioned in previous chapters, one must bear in mind that structural subtyping is not always the appropriate solution; there do exist situations in which nominal subtyping is more appropriate (Sect. 1.3).

To my knowledge, this is the first systematic corpus analysis to determine the benefits of structural subtyping. The contribution of this chapter are: (1) identification of a number of characteristics in a program that suggest the use of implicit structural types; and (2) results from automated and manual analyses that measure the identified characteristics.

4.2 Corpus and Methodology

For this study, the source code of up to 29 open-source Java applications were examined (version numbers of the applications are provided in Appendix B.1). The full set of subject programs were used for the automated analyses, while (while for practical considerations) manual analyses were performed on various subsets of these (ranging from 2 to 8 members). The applications were chosen from the following sources: popular applications on SourceForge, Apache Foundation applications, and the DaCapo benchmark suite.[3]

The full set of programs range from 12 kLOC to 161 kLOC, programs that were selected based on size, type (library/framework vs. sealed applications[4]) and domain (selecting for variety). For some of the manual analyses, I favored applications with which I was familiar (as this aided analysis), but I also aimed for variety in both application type and domain. All of the manual analyses, including the subjective analyses, were performed by one observer only—the author. The methodology for each analysis is described in the corresponding section; further details are available in Appendix B.2.

4.3 Inferring Structural Types for Method Parameters

It is considered good programming practice to make parameters as general as the program allows. Bloch, for example, recommends favoring interfaces over classes in general—particularly so in the case of parameter types [Bloch 2001]. An analogous situation arises in the *generic programming* community, where it is recommended that generic algorithms and types place as

[3] http://dacapobench.org/

[4] Here I define a *sealed application* as a complete program that is not intended to be directly reused.

few requirements as possible on their type parameters (e.g., what methods they should support) [Musser and Stepanov 1989].

Bloch acknowledges that sometimes an appropriate interface does not exist. For example, class java.util.Random implements only one (empty) marker interface. In such a case the programmer is forced to use classes for parameter types—even though it is possible that multiple implementations of the same functionality could exist [Bloch 2001]. This is a situation where structural subtyping could be beneficial, as it allows programmers to create supertypes after-the-fact.

As it is impossible to retroactively implement interfaces in Java, I hypothesized that method parameter types are often overly specific, and sought to determine both (1) the degree and (2) the character of over-specificity. To answer question (1), an automated whole-program analysis to infer structural types for method parameters was performed. Methodology and quantitative results are described in Sect. 4.3.1. To properly interpret this data, however, we must consider question (2). Accordingly, the inferred structural types from the previous analysis were manually examined and the following qualitative question was considered: would changing a method to have the most general structural type potentially improve the method's interface (Sect. 4.3.2)? Across all applications, the occurrences of inferred structural types that were supertypes of classes and interfaces of the Java Collections Library were enumerated. Of these, in Sect. 4.3.3 those structural types that a client might plausibly wish to implement while *not* simultaneously implementing a more specific nominal type (e.g., Collection, Map, etc.) are presented.

4.3.1 Quantitative Results

The analysis infers structural types for method parameters, based on the methods that were actually called on the parameters. (For example, a method may take a List as an argument, but may only use the add and iterator methods.) The analysis, a simple inter-procedural dataflow analysis, re-computes structural types for each parameter of a method until a fixpoint is reached. Structural types were not inferred for calls to library methods (for modularity purposes), nor were they inferred for primitive types, common types such as String and Object, and cases where the inferred structural type would have a non-public member. Finally, to simplify the analysis, structural types were *not* inferred for objects on the left-hand side of an assignment expression.

The analysis is conservative; in the case where a parameter is not used (or only methods of class Object are used), no structural type is inferred for it. A parameter may be unused because (a) it is expected that overriding methods will use the parameter, or (b) because the method may make use of the parameter when the program evolves, or (c) because it is no longer needed, due to changes in the program. In the case of method overriding, the analysis ensures that the same structural types are inferred for corresponding parameters in the entire method family.

The first set of results appear in Table 4.1. In Ant, 9.7% of parameters were unused, 47.2% of parameters had a primitive type, were String, or were Object. For 0.3% of parameters, a call was made to a non-public method, which means that a structural type could not be used in this case (as the visibility of all members of a structural interface must be public). 1.0% of parameters could not have a structural type inferred due to the fact that the associated method was overriding a method in a library. Finally, for 15.8% of parameters, a structural type could not be inferred, due

	LOC	Unused	Primitive type	Non-public call	Library override	Called library	Assigned	% Inferrable of total	% Inferrable of candidates
Ant	62k	9.7%	47.2%	0.3%	1.0%	9.1%	15.8%	17.0%	40.5%
antlr	42k	16.7%	50.2%	0.4%	0.2%	8.1%	9.0%	15.5%	47.6%
Apache coll	26k	8.4%	55.0%	2.1%	6.0%	1.2%	16.4%	11.0%	38.4%
Areca	35k	9.7%	39.4%	0.1%	4.8%	9.3%	20.5%	16.1%	35.1%
Cayenne	95k	8.3%	47.0%	0.5%	3.1%	8.1%	11.2%	21.9%	53.2%
Columba	70k	11.6%	40.8%	0.6%	19.8%	5.5%	9.0%	12.6%	46.4%
Crystal	12k	18.0%	4.1%	0.2%	17.9%	7.4%	22.3%	30.1%	50.3%
DrJava	59k	13.5%	42.5%	0.8%	13.2%	7.8%	7.8%	14.3%	47.9%
Emma	23k	20.5%	42.3%	0.4%	0.9%	7.4%	8.8%	19.6%	54.6%
freecol	62k	8.7%	38.5%	0.0%	11.5%	3.9%	11.8%	25.5%	61.9%
hsqldb	62k	14.4%	61.3%	6.4%	3.9%	1.0%	4.8%	8.2%	58.5%
HttpClient	18k	14.3%	55.7%	0.1%	0.3%	5.7%	5.1%	18.8%	63.5%
jEdit	71k	11.8%	56.9%	1.0%	9.7%	4.9%	8.5%	7.2%	35.1%
JFreeChart	93k	8.1%	45.9%	0.4%	1.4%	14.3%	10.4%	19.6%	44.2%
JHotDraw	52k	18.3%	32.3%	0.0%	7.7%	11.1%	10.1%	20.5%	49.2%
jruby	86k	19.7%	27.2%	0.4%	0.8%	3.0%	16.0%	32.9%	63.4%
jung	26k	8.1%	33.8%	0.1%	4.8%	22.9%	12.0%	18.2%	34.3%
LimeWire	97k	13.7%	45.8%	1.4%	7.1%	7.9%	6.7%	17.5%	54.5%
log4j	13k	12.3%	46.8%	0.7%	4.7%	6.6%	10.0%	18.8%	53.1%
Lucene	24k	12.3%	58.3%	0.6%	0.1%	4.9%	14.8%	9.2%	31.8%
OpenFire	90k	14.0%	39.7%	0.2%	6.1%	7.6%	10.9%	21.4%	53.5%
plt collections	19k	15.8%	19.5%	0.3%	3.0%	6.8%	42.1%	12.4%	20.3%
pmd	38k	31.3%	32.7%	0.0%	1.3%	6.5%	8.4%	19.7%	56.9%
poi	50k	15.9%	69.8%	0.7%	3.3%	1.3%	2.1%	6.7%	66.2%
quartz	22k	15.4%	54.2%	0.0%	0.8%	5.9%	5.6%	18.2%	61.2%
Smack	40k	17.2%	45.3%	0.2%	1.6%	12.5%	8.1%	15.1%	42.2%
Struts	28k	6.3%	58.1%	0.1%	4.4%	5.1%	18.9%	7.1%	22.8%
Tomcat	126k	13.6%	54.6%	0.1%	3.2%	3.7%	11.0%	13.8%	48.3%
xalan	161k	10.5%	56.5%	1.3%	2.7%	2.5%	10.9%	15.7%	54.1%
Average		**13.7%**	**44.9%**	**0.7%**	**5.0%**	**7.0%**	**12.0%**	**16.7%**	**47.9%**

Table 4.1: Categories of method parameters when running structural type inference over 29 programs. "Unused" denotes the percentage of parameters that were not transitively used in the program, "primitive type" is the percentage of parameters that were either a primitive type, or were String or Object, "non-public call" is the percentage of parameters on which a non-public method was called (in which case a structural type could not be inferred), and "library override" is the percentage of paramters for which a structural type could not be inferred due to the fact that the method was an override of a library method. "Called library" is the percentage of parameters for which a structural type could not be inferred because a library method was transitively called and "assigned" is the percentage of parameters that were assigned to a local or member variable and did not have structural types computed. "Percent inferrable of total" is the percentage of all parameters that could have a structural type inferred, while "percent inferrable of candidates" is the percentage of inferrable parameters, when considering only those parameters for which a structural type would be meaninful.

to the fact that the parameter was assigned to a local variable or member variable (this was a limitation of the analysis).

Considering all parameters, an average of 16.7% could have a structural type inferred. However, if we exclude parameters that fall into the categories in columns 3–6 (i.e., unused parameters, primitive types, non-public calls, and library overrides), then an average of 47.9% of parameters could have a structural type inferred. This second figure is more relevant, as it is not meaningful to infer structural types for parameters that fall into the aforementioned categories.

The analysis also computed some characteristics of these structural types that were inferred; results are displayed in Table 4.2. An average of 94.0% of parameters were declared with an overly precise nominal type (i.e., the nominal type contained more methods than were actually needed). For an average of 91.8% of the inferred parameters a corresponding nominal type did *not* exist in the program that would make the parameter type as *general* as possible (i.e., a nominal type that contained only those methods transitively called on the object). There were an average of 3.7 methods in the inferred structural types, across all programs, while there were an average of 41.7 methods in the corresponding nominal types. Finally, there was an average median of 1.2 structural types inferred for each nominal type in the program, and an average maximum of 23.4 structural types.

Note that the data shows that inferred structural types do not have many methods,[5], while the corresponding nominal types have quite a few methods. This shows that there is quite a large *degree* of over specificity—more than a full order of magnitude—in addition to the large percentage of overly specific parameters. This is likely due to the overhead of naming and defining nominal types, as well as the lack of retroactive interface implementation. The analysis also showed that when nominal types were as general as possible, they had very few members—one or two on average. This is in accordance with previous work which found that interfaces are generally smaller than classes [Tempero et al. 2008].

Additional data. For a given nominal type, there were not many corresponding structural types (2.5 on average, a median of 1.2). The data followed a power law distribution, with an average maximum of 24; that is, small values were heavily represented, but there were also a few large values. The low median suggests that the overhead of naming structural types is not necessarily high; it is plausible that programmers would be able to name and use structural types for around half of the nominal parameter types.

Finally, if we were to define new interfaces everywhere possible, the average increase in the number of interfaces is 313%, the median is 287%, and the maximum is 1000%. This illustrates the infeasibility of defining new nominal types for the inferred structural types. Note that only those interfaces for which the implements clause of a class could be modified (i.e., those classes in the program's source) were considered; in general, the situation is even worse, as programmers may wish to define new supertypes for types contained in libraries.

[5] There is one outlier in the data; in pmd, inferred structural types had 29.5 methods on average. This is due to the use of the visitor design pattern—all visit methods are accessible from the top visitor accept method, since each override calls a specific visit method.

	LOC	% Inferrable	% Overly	% Structural	Avg methods/	Avg methods/	Struct types/nominal	
			specific	needed	structural type	nominal type	median	max
Ant	62k	40.5%	98.6%	97.7%	2.1	36.0	1	27
antlr	42k	47.6%	100.0%	98.8%	2.2	14.0	2	9
Apache coll	26k	38.4%	89.5%	83.0%	2.0	16.0	1	11
Areca	35k	35.1%	99.1%	97.4%	2.8	35.9	1	35
Cayenne	95k	53.2%	96.3%	92.6%	2.4	31.3	2	27
Columba	70k	46.4%	99.6%	98.8%	1.9	51.6	1	19
Crystal	12k	50.3%	98.8%	96.6%	3.2	15.7	1	19
DrJava	59k	47.9%	89.5%	87.1%	3.2	56.3	1	20
Emma	23k	54.6%	88.2%	87.8%	3.4	17.1	1	9
freecol	62k	61.9%	98.6%	97.9%	2.6	84.8	1	57
hsqldb	62k	58.5%	99.4%	99.4%	1.7	48.9	2	34
HttpClient	18k	63.5%	96.3%	94.8%	3.5	27.0	1	17
jEdit	71k	35.1%	95.5%	95.5%	2.2	119.6	1	20
JFreeChart	93k	44.2%	97.7%	93.5%	3.3	53.9	1	35
JHotDraw	52k	49.2%	100.0%	97.2%	3.0	57.0	2	19
jruby	86k	63.4%	98.1%	97.5%	6.9	66.1	1	85
jung	26k	34.3%	96.3%	88.3%	1.8	32.1	1	15
LimeWire	97k	54.5%	98.5%	94.9%	2.1	34.6	1	21
log4j	13k	53.1%	96.5%	95.0%	2.3	56.7	1	6
Lucene	24k	31.8%	80.5%	77.4%	1.6	13.5	1.5	8
OpenFire	90k	53.5%	99.4%	96.7%	2.4	37.1	1	45
plt collections	19k	20.3%	58.2%	59.3%	1.5	39.8	1	25
pmd	38k	56.9%	72.9%	69.1%	29.5	48.2	2	23
poi	50k	66.2%	88.0%	87.0%	1.9	22.8	1	8
quartz	22k	61.2%	100.0%	99.1%	2.2	36.6	1	11
Smack	40k	42.2%	100.0%	91.6%	4.4	29.2	1	13
Struts	28k	22.8%	96.4%	96.4%	2.1	32.4	1	13
Tomcat	126k	48.3%	96.8%	96.3%	4.5	37.6	2	32
xalan	161k	54.1%	96.4%	96.0%	5.3	56.5	1	16
Average		**47.9%**	**94.0%**	**91.8%**	**3.7**	**41.7**	**1.2**	**23.4**

Table 4.2: Results of running structural type inference. *Percent inferrable* is the percentage of candidate parameters that could have a structural type inferred (i.e., last column in Fig. 4.1), *percent overly specific* is the percentage of the inferrable parameters that have an overly specific nominal type, *percent structural needed* is the percentage of the inferrable parameters for which a most general nominal type does *not* exist, *average methods per structural type* is the average number of methods in the inferred structural types, *average methods per nominal type* is the average number of methods in nominal types that appear as parameter types (including inherited methods), and *median/maximum structural types per nominal* are the median and maximum, respectively, of the number of inferred structural types corresponding to each nominal type.

4.3.2 Qualitative Results

Though the results show that many parameters are overly specific, it is not necessarily a good design to make every parameter as general as possible. This is because a method might be currently only using a particular set of methods, but later code modifications may make it necessary to use a larger set; a more general type could hinder program evolution. On the other hand, more general types make methods more reusable, which aids program evolution. For this reason, a refactoring to structural types (or even structural type inference) cannot be a fully automated process—programmers must consider each type carefully, keeping in view the kinds of program modifications that are likely to occur. Additionally, for some structural types, there may ever be only one corresponding nominal type, in which case using a structural type is of limited utility.

Accordingly, an empirical question was considered: would changing a given method to have the most general structural types for its parameters make the method more general in a way that could improve the program? To determine this, I inspected each method and asked two questions. First, does the inferred parameter type *S* generalize the abstract operation performed by the method, as determined by the method name? Second, does it seem likely that there would be multiple subtypes of *S*?

Two applications were studied: Apache Collections (a collections library) and Crystal (a static analysis framework). Of methods for which a structural type was inferred on one or more parameters, I found that 58% and 66%, respectively, would be generalized in a potentially useful manner if the inferred types were used.

For example, in Apache Collections, in the class OnePredicate (a predicate class that returns true only if one of its enclosing predicates returns true), the factory method getInstance(Collection) had the structural type { iterator(); size(); } inferred for its parameter. This would make the method applicable to any collection that supported only iteration and retrieving the collection size, even if it didn't support collection addition and removal methods. There were 25 other methods in the library that used this structural type. Another example is the method ListUtils.intersection which takes two List objects. However, the first List need only have a contains method, and the second List need only have an iterator method (for this latter parameter, the interface Iterable could be used). There were also 8 methods that took an Iterator as a parameter, but never called the remove method. With a structural type for the method, the type would clearly specify that a read-only iterator can be passed as an argument.

In Crystal, two methods took a Map parameter that used only the get and put methods. Converting the method to use this structural type would make it applicable to a map that did not support iteration (such a type exists in Apache Collections, for example). Also, there were 11 methods that use only the methods getModifiers() and getName() on an IBinding object (an interface in the Eclipse JDT). Replacing the nominal type with a structural type would allow the program to substitute a different "bindings" class that supported only those two methods.

Of course, for some of these structural types, there may not be a large number of classes that implement its methods but not all of the methods of a more specific nominal type, e.g., Collection. However, I believe that all of the aforementioned types represent meaningful abstractions. Furthermore, since it is conceivable that a programmer may define a class implementing that abstraction, using these more general types would increase the applications' reusability.

Translation to Whiteoak

Using the inference algorithm, I also developed an automated translation of programs from Java to Whiteoak [Gil and Maman 2008], a research language that extends Java with support for structural subtyping. I performed this translation on two programs: Apache Collections and Lucene, confirming the correctness of the analysis and demonstrating its practical use.

Methods in type	Uses	Description
`get(Object); containsKey(Object);`	168	Read-only non-iterable map; for instance, a read-only hashtable[6]
`iterator(); isEmpty(); size();`	114	Read-only iterable collection that knows its size; for instance, a read-only list
`add(Object); addAll(Collection);`	101	Write-only collection; for instance, a log
`put(Object, Object);`	55	Write-only map
`hasNext(); next();`	28	Read-only iterator
`contains(Object);`	21	Read-only collection that does not support iteration; for instance, a read-only hashset
`get(Object); put(Object, Object);`	15	Non-iterable map; for instance, a hashtable
`contains(Object); iterator(); size();`	11	Read-only iterable collection that knows its size and can be polled for the existence of an element; for instance, an iterable hashset
`add(Object); contains(Object); iterator(); size();`	10	Same as above, but that also supports adding elements
`iterator(); size(); toArray(Object[]);`	8	Read-only collection that can be converted to an array; for instance, a read-only array

Table 4.3: Uses of Java Collections classes across 29 programs, as inferred using the parameter structural type inference. (Erasures are used in lieu of generic types.)

4.3.3 Uses of Java Collections Library

I next considered inferred structural types that were supertypes of interfaces and classes in the Java Collections Library. Over all applications, there were 67 distinct types in total, though not all appeared to express an important abstraction. I made a conservative subjective finding that at least 10 of these types *were* potentially useful; these are displayed in Table 4.3, along with a description of possible implementations. For instance, there were 168 inferred parameters that used only the get() and containsKey() methods of Map. It would be useful to have a type corresponding to this abstraction, particularly if the map is immutable and must have its contents set at creation-time. A type consisting of these two methods would also be useful to support the pattern that once a map is populated, clients should not make modifications.

The relatively high number of occurrences of each of these structural types suggests their utility, even though the types contain few methods. It further shows that programs routinely make use of types that the library designers either did not anticipate or chose not to support.

In summary, the data shows that programs make repeated use of many implicit structural types. A language that would allow defining these types explicitly could be beneficial, as it can help programmers make their methods more generally applicable.

4.3.4 Related work

Forster [Forster 2006] and Steimann [Steimann 2007] have described experience using the Infer Type refactoring, which generates new interfaces for inferred types and replaces uses of overly specific types with these interfaces. This analysis is more general than the one used here, because it considers all type references, not just parameter types. However, the refactoring is limited by the fact that classes in libraries cannot retroactively implement new interfaces. Steimann found that when applying this refactoring, the number of total interfaces almost quadrupled—an increase of 369%.[7] In his analysis, there were an average of 2.8 and 4.5 reference per inferred type, respectively, in the two studied applications, DrawSWF and JHotDraw. By comparison, in DrawSWF, I found that structural types were used in an average of 2.7 parameters; for JHotDraw this value was 3.3, which differs from Steimann's result. This discrepancy is likely due to the fact that he considered types other than those of parameters. Additionally, both Forster and Steimann found that the number of variables typed with each new inferred interface followed a power law distribution, which is what I also found for parameters.

Summary of results

In summary, the parameter analysis suggests that there are many nominal types that could be made more general using structural subtyping, and most of these were qualitatively determined to be useful. Also, the inferred structural types had an order of magnitude fewer methods than the corresponding nominal types. It is infeasible to define new nominal types to correct this, due to the number of structural types inferred per nominal type and the resulting percentage increase in interfaces.

4.4 Throwing "Unsupported Operation" Exceptions

In the Java Collections Library, there are a number of "optional" methods whose documentation permits them to always throw an exception. This decision was due to the practical consideration of avoiding an "explosion" of interfaces; the library designers mentioned that at least 25 new interfaces would be otherwise required [Sun Microsystems 2003].

To determine if such super-interfaces would be useful in practice, the methods in the subject programs that unconditionally throw an UnsupportedOperationException were totalled. The program that had the most such methods was Apache Collections: there were 148 methods that unconditionally throw the exception (out of 3669 total methods, corresponding to 4%). Next, those methods that were overriding a method in the Java Collections Library were considered. To encode these optional methods directly would require 18 additional interfaces. There are only 27 interfaces defined in the library, so this represents a 67% increase. Note that this is a conservative estimate, as interactions between classes (e.g., an Iterable returning a read-only Iterator) were not considered. A selection of these structural super-interfaces is summarized in Table 4.4. For instance, there were 50 iterator classes that did not support the remove() operation, and 19 subclasses of Collection that supported a read-only interface.

Note that, with the exception of the read-only iterator, the sets of interfaces in Tables 4.4 and 4.3 are distinct from one another (though some are subtypes). This is likely due to the fact that different applications use different subsets of the methods of a class.

[7] This differs slightly from our average of 313%, though this difference is likely due to the fact that Steimann considered only two applications.

	Number of classes
Read-only `Iterator`	50
Read-only `Collection`	19
Read-only `Map`	9
Read-only `Map.Entry`	6
Read-only `ListIterator`	6
`Collection` supporting everything but removal	5
`Map` supporting everything but removal	4
`Collection` supporting only read and removal methods	1
`Collection` supporting iteration, addition, and size only	1
`ListIterator` supporting read, add, and remove (but not `set()`)	1
`ListIterator` supporting only read and `set()` operation	1
`Map` supporting read, put, and size only	1
`Map` supporting read and put, but not size or removal	1
`Map` supporting everything but `entrySet()`, `values()` and `containsValue()`	1

Table 4.4: A selection of the structural interfaces "implemented" by classes in the subject programs once methods unconditionally throwing an UnsupportedOperationException are removed. (Actual method sets are omitted to conserve space.)

Structural subtyping could be helpful for statically ensuring that "unsupported operation" exceptions cannot occur, as it would allow programmers to express these super-interfaces directly.

4.5 Common Methods

In my experience, there are situations where two types share an implicit common supertype, but this relationship is not encoded in the type hierarchy. For example, suppose two classes both have a getName method with the same signature, but there does not exist a supertype of both classes containing this method. I call getName, and methods like it, *common methods*. Common methods can occur when programmers do not anticipate the utility of a shared supertype or when two methods have the same name, but perform different operations; e.g., Cowboy.draw() and Circle.draw() [Magnusson 1991].

Accordingly, this section aims to answer three questions: (1) how often do common methods occur, (2) how many common methods represent an accidental name clash, and (3) do common methods result in code clones?

4.5.1 Frequency

A simple whole-program analysis to count the number of common methods in each application was performed. Only public instance methods were considered (resulting in slightly different data than that previously presented [Malayeri and Aldrich 2008a]). Results are in Table 4.5. Overall, common methods comprise an average of 19% of all public instance methods. That is, for 19% of methods, there existed another method with the same name and signature and the method was not contained in a common supertype of the enclosing types.

	LOC	Number of types	Types with >1 common method	Percentage	% common methods	Avg # classes/ common signature
Ant	62k	945	65	6.9%	31.3%	3.7
antlr	42k	226	26	11.5%	23.6%	2.7
Apache Collections	26k	550	19	3.5%	7.3%	2.7
Areca	35k	362	30	8.3%	15.4%	2.7
Cayenne	95k	1415	104	7.3%	18.1%	2.8
Columba	70k	1232	48	3.9%	17.3%	3.1
Crystal	12k	211	4	1.9%	5.1%	2.9
DrJava	59k	927	65	7.0%	12.1%	2.6
Emma	23k	443	22	5.0%	18.7%	3.4
freecol	62k	569	55	9.7%	20.6%	2.7
hsqldb	62k	355	31	8.7%	19.5%	2.6
HttpClient	18k	231	19	8.2%	15.0%	2.6
jEdit	71k	880	40	4.5%	11.7%	2.5
JFreeChart	93k	789	301	38.1%	39.5%	3.9
JHotDraw	52k	616	59	9.6%	19.0%	2.8
jruby	86k	997	83	8.3%	15.6%	3.1
jung	26k	531	24	4.5%	19.3%	2.4
LimeWire	97k	1689	88	5.2%	17.7%	3.1
log4j	13k	201	4	2.0%	13.6%	2.4
Lucene	24k	398	21	5.3%	13.4%	2.6
OpenFire	90k	1039	110	10.6%	19.0%	3.0
plt collections	19k	812	60	7.4%	7.5%	2.8
pmd	38k	478	24	5.0%	12.0%	2.7
poi	50k	539	62	11.5%	20.9%	2.6
quartz	22k	158	24	15.2%	20.0%	2.4
Smack	40k	847	115	13.6%	23.5%	3.3
Struts	28k	609	158	25.9%	45.2%	2.7
Tomcat	126k	1727	234	13.5%	32.6%	3.6
xalan	161k	1223	94	7.7%	16.1%	2.9
Average				**9.3%**	**19.0%**	**2.9**

Table 4.5: Common methods for each application. *Number of types* indicates the total number of types in the application, *types with greater than one common method* is the number of types that share more than one common method, *percentage* is the percentage of this compared to the total number of types, *percent common methods* is the percentage of public instance methods that is a common method, and *average number of classes per common signature* is the average number of classes for each common method signature.

The number of types that share at least two common methods with another type was also computed; there were an average of 9% of such types. These are the cases in which a structural supertype is most likely to be useful. This high percentage indicates that there are a number of implicit structural types in most applications.

For example, in Apache Collections, UnmodifiableSortedMap and OrderedMap share the methods firstKey() and lastKey(). And, AbstractLinkedList and SequencedHashMap share the methods getFirst() and getLast(). Finally, BoundedMap and BoundedCollection have the common methods isFull() and maxSize().

In Lucene, a document indexing and search library, RAMOutputStream and RAMInputStream both support the seek(), close(), and getFilePointer() methods, which might be useful to move to a supertype. Also, the classes PhraseQuery and MultiPhraseQuery both support the methods add(Term), getPositions(), getSlop(), and setSlop(int).

4.5.2 Accidental Name Clashes

Of course, to interpret this data, we must consider cases where the common methods have the same *meaning*, and where callers are likely to call the methods with the same purpose in mind. If two methods have the same meaning, it might be useful to define a structural type consisting of that method. Two methods are defined as "having the same meaning" if they perform the same abstract operation, taking into account (a) the semantics of the method, and (b) the semantics of the enclosing types. This determination was made by examining the source code, using javadoc where available.

Two applications were studied: Apache Collections and Lucene. In Collections, under condition (a), there were no methods that had the same signature but performed different abstract operations. However, there were 2 cases (1% of all common methods) where the methods had the same meaning, but the enclosing classes did not appear to be semantic subtypes of some common supertype containing that method; i.e., condition (b) was not satisfied. For example, the classes ChainedClosure and SwitchClosure both had a getClosures() method, but ChainedClosure calls each of these closures in turn, while SwitchClosure calls that closure whose predicate returns true.

In Lucene, there were 42 instances of methods that had the same signature, but did not have the same meaning (19% of all common methods). In 32 of these cases, the methods were actually performing a different abstract operation. For example, HitIterator.length() returned the number of hits for a particular query, while Payload.length() returned the length of the payload data. An additional 10 cases did not satisfy condition (b) above. For example, in a high-level class IndexModifier, there were several cases where a method m performed some operation, then called IndexWriter.m, the latter performing a lower-level operation. So, the semantics of the methods were similar, but the semantics of each class was different.

Overall, the data is promising, as it indicates that most common methods have the same meaning and would benefit from being contained in a structural supertype—90% on average, across both applications. Structural subtyping would allow these methods to be called in a generic manner, without the need to create additional interfaces.

4.5.3 Code Clones

I hypothesized that common methods can lead to code clones, as there is a common structure that is not expressed in the type system. To determine this, two applications were examined: Eclipse JDT and Azureus.

In the Eclipse Java Development Tools (JDT), the classes FieldAccess and SuperFieldAccess have no superclass other than Expression. The same problem occurs with MethodInvocation and SuperMethodInvocation, and ConstructorInvocation and SuperConstructorInvocation. I found 44 code clones involving these types (though some were only a few lines long). An example of a code clone involving MethodInvocation and SuperMethodInvocation appears in Fig. 4.1. Another code clone involving SuperConstructorInvocation and ConstructorInvocation appears in Fig. 4.2.

```
private InlineMethodRefactoring(ICompilationUnit unit,
 MethodInvocation node, int offset, int length)
{
   this(unit, (ASTNode)node, offset, length);
   fTargetProvider= TargetProvider.create(unit, node);
   fInitialMode= fCurrentMode= Mode.INLINE_SINGLE;
   fDeleteSource= false;
}

private InlineMethodRefactoring(ICompilationUnit unit,
 SuperMethodInvocation node, int offset, int length)
{
   ...  // same method body as above
}
```

Figure 4.1: Example of code duplication in the Eclipse JDT. Structural subtyping could eliminate this duplication.

Similarly, in the Eclipse SWT (Simple Windowing Toolkit), there are 13 classes (such as Button, Label, and Link) with the methods getText and setText that get and set the main text for the control. But, there is no common IText interface. Azureus, a BitTorrent client, is an application that requires the ability to call these methods in a generic fashion. Azureus is localized for a number of languages, which can be changed at runtime. Accordingly, there are several instances of code similar to that of Fig. 4.3.

Note that some of this code duplication might be avoided if the class hierarchy were refactored. Obviously, this is not always possible—e.g., Azureus cannot modify SWT.

The code duplication in these examples can be dramatically reduced by taking advantage of structural type. For example, Fig. 4.4 shows how the code block of Fig. 4.3a could be re-written with Unity structural types.

In summary, common methods can lead to undesirable code duplication. Structural subtyping can help eliminate this problem, without refactoring the class hierarchy.

4.6 Cascading instanceof Tests

I considered the question of whether structural subtyping could provide benefits if used in conjunction with other language features—external methods in particular.

Since Java does not support any form of external dispatch, programmers often compensate by using cascading instanceof tests in client code. This programming pattern is problematic because it is tedious, error-prone, and lacks extensibility [Clifton et al. 2006]. Many instances of this pattern could be re-written to use external methods, but a problem arises if an instanceof test is performed on an expression of type Object.

```
case ASTNode.SUPER_CONSTRUCTOR_INVOCATION: {
   SuperConstructorInvocation superInvocation= (SuperConstructorInvocation) parent;
   IMethodBinding superBinding= superInvocation.resolveConstructorBinding();
   if (superBinding != null) {
      return getParameterTypeBinding(node, superInvocation.arguments(), superBinding);
   }
   break;
}
case ASTNode.CONSTRUCTOR_INVOCATION: {
   ConstructorInvocation constrInvocation= (ConstructorInvocation) parent;
   IMethodBinding constrBinding= constrInvocation.resolveConstructorBinding();
   if (constrBinding != null) {
      return getParameterTypeBinding(node, constrInvocation.arguments(), constrBinding);
   }
   break;
}
```

Figure 4.2: Code duplication involving SuperConstructorInvocation and ConstructorInvocation. Only the highlighted lines of code differ in the two blocks.

To illustrate this, let us consider how instanceof tests would be translated to external methods. Suppose we have a cascaded instanceof, with each case of the form "[else] if expr instanceof C_i { $block_i$ }." This would be translated to an external method f defined on expr's class, and overridden for each C_i by defining $C_i.f$ { $block_i$ }. The top part of Fig. 4.5b shows the external methods translated from the instanceof tests in Fig. 4.5a (but without an external method defined on Object, the type of query, which I will come to in a moment).

A problem arises when the target expression in the instanceof test is of type Object, as an external method must be defined on Object, then overridden for each type tested via an instanceof. The problem with this solution is that it pollutes the interface of Object. In many cases, the implementation of this method performs a generic fallback operation that does not make sense for an object of arbitrary type—but this method becomes part of every class's interface and implementation. (While it is also possible to pollute the interface of an arbitrary class C, this is generally less severe, and detecting such a situation requires application-specific knowledge.)

To determine the prevalence of this pattern, instanceof tests in 8 applications were manually examined. The result of this analysis was that 13% to 54% (with an average of 26%) were performing a cascading instanceof test on an expression of type Object (see Table 4.6).

Structural subtyping provides one solution to this problem. The type of the expression on which the instanceof is performed would be changed from Object to the structural type consisting of the newly defined external method f. That is, instead of making the target operation applicable to an arbitrary object, it would be applicable to only those objects that contain method f. Figure 4.5b defines an external method toQuery on String and Query, then uses the structural type { toQuery(…) } as the type for the List elements. The advantage of using structural subtyping

```
if (widget instanceof Label)
   ((Label) widget). setText(message);
else if (widget instanceof CLabel)
   ((CLabel) widget). setText(message);
else if (widget instanceof Group)
   ((Group) widget). setText(message);
... // 5 more items
```

(a)

```
if (widget instanceof CoolBar) {
   CoolItem[] items = ((CoolBar)widget).getItems();
   for(int i = 0; i < items.length; i++) {
      Control control = items[i].getControl();
      updateLanguageForControl(control);
   }
} else if (widget instanceof TabFolder) {
   ...  // same code as highlighted above
} else if (widget instanceof CTabFolder) {
   ...  // same code as highlighted above
... // 5 more items
```

(b)

Figure 4.3: Code excerpts from Azureus, illustrating an awkward coding style and duplication.

```
let
    widget: Widget(setText: () ⇒ string → unit) = . . .
in
    widget.setText message
```

Figure 4.4: Code block of Fig. 4.3a re-written in Unity.

is that the main code can call this method uniformly.[8]

Thus, for many applications, there is a potential benefit to using structural subtyping in a language that supports external dispatch; an average of 26% of instanceof tests could be eliminated.

Note that since we refined the element type of the List object, this obviates the need for the error condition—an additional advantage. However, it is not always possible to refine types to a structural type; an expression may simply have type Object, due to the loss of type information. In such a case, it would be possible to re-write the code using a structural downcast. Though the use of casts would not be eliminated, there are still several advantages to this implementation style. First, the external methods could be changed without having to also modify the method that uses them. Also, if subclasses are added, a new internal or external method could be defined for them. Finally, since the proposed cast would use a structural type, it would be more general, applying to any type for which the method were defined.

[8] Note that it would not be possible to make use of a nominal interface containing the method f to call the method in a generic manner. Recall that for external methods to be modular, once a method is defined as an internal method, it cannot be implemented with an external method; see Sect. 2.4.2.

Original Java Code

```
List qlist = ...
Object query = qlist.get(i);
Query q = null;
if (query instanceof String)
   q = parser.parse((String) query);
else if (query instanceof Query)
   q = (Query) query;
else
   System.err.println("Unsupported query type");
```

Unity Re-Write

```
method string.toQuery: ( ) ⇒ QueryParser → Query =
    fun parser: QueryParser --> parser.parse this
method Query.toQuery: ( ) ⇒ QueryParser → Query =
    fun _ --> this

. . .
using toQuery in
    type QueryConvert = Object(toQuery: ( ) ⇒ QueryParser → Query)
    List<QueryConvert> qlist = . . .
    let q : Query = qlist.get(i).toQuery(parser)
```

Figure 4.5: Rewriting instanceof using structural subtyping and external dispatch. At the top is the original code; below is the translated code, which defines the structural type QueryConvert and external methods on Query and String. Note that the translated code eliminates the need for the error condition.

4.7 Java Reflection Analysis

I aimed to answer the following question: do Java programs use reflection where structural types would be more appropriate? Uses of reflection likely fall into two categories: cases where dynamic class instantiation and classloading are used, and cases where the type system is not sufficiently powerful to express the programming pattern used. It is difficult to eliminate reflection in the first category, as these uses represent an inherently dynamic operation. However, some of the uses in the second category could potentially be rewritten using structural downcasts. Reducing the uses of reflection is beneficial as it decreases the number of runtime errors and can improve performance.

Across the 29 subject programs, an average of 32% of uses of the reflection method Class.getMethod could be re-written using a structural downcast (see Table 4.7). A structural downcast is preferable to reflection because type information is retained when later calling

	instanceof	Expression of type Object	Percentage
Apache collections	225	75	33%
Areca	77	10	13%
JHotDraw	229	50	22%
log4j	54	8	15%
Lucene	56	10	18%
PLT collections	119	64	54%
Smack	56	20	36%
Tomcat	959	158	16%
Average			**26%**

Table 4.6: Total instanceof tests, the number present in cascading if statements that perform the test on an expression of type Object, and that number expressed as a percentage. Code written using this pattern can be translated to a language with structural subtyping and external dispatch.

methods, as opposed to Method.invoke, which is passed an Object array and must typecheck the arguments at runtime. Additionally, it is easier to combine sets of methods in a downcast; when using reflection, each method must be selected individually. There is also the potential to make method calls more efficient, which is difficult with reflection, due to the low-level nature of the available operations. (For example, the language Whiteoak [Gil and Maman 2008] supports efficient structural downcasts.)

In summary, the high percentage of reflection uses that can be translated to structural downcasts suggests that programmers may sometimes use reflection as a workaround for lack of structural types.

4.8 Summary and Conclusions

In summary, I found that a number of different aspects of Java programs suggest the potential utility of structural subtyping. While some of the results are not as strong as others, taken together the data suggests that programs could benefit from the addition of structural subtyping, even if they were written in a nominally-typed language. In particular, structural subtyping has the potential be used to improve the reusability and maintainability of existing object-oriented programs.

This chapter provided evidence to support hypothesis II. In particular, I presented evidence suggesting that structural subtyping could help make method parameters more general (Sect. 4.3). There was a high frequency of common methods (Sect 4.5.1), and a low frequency of common methods representing an accidental name clash (Sect 4.5.2). Finally, we saw evidence that some cases of code duplication could be avoided with structural subtyping (Sect. 4.5.3).

Additionally, I showed that existing language designs can lead to coding patterns that defer errors to runtime, coding patterns that could be re-written with structural subtyping to provide more static typechecking in these situations. In particular, some Java runtime exceptions (i.e., OperationUnsupportedException) can be eliminated in a straightforward manner with a de-

	Uses of getMethod( )	Could be rewritten	Percentage
Ant	36	9	25%
Apache Collections	4	3	75%
Areca	1	0	0%
Azureus	27	6	22%
Cayenne	28	4	14%
columba	10	8	80%
DrJava	7	2	29%
emma	2	1	50%
freecol	1	1	100%
hsqldb	2	0	0%
HttpClient	8	6	75%
jedit	10	7	70%
jfreechart	1	1	100%
JHotDraw	26	1	4%
jruby	17	6	35%
jung	1	1	100%
log4j	4	1	25%
openfire	2	0	0%
pmd	2	2	100%
quartz	3	2	67%
struts	2	0	0%
tomcat	37	10	27%
xalan	28	11	39%
Totals	**259**	**82**	**32%**

Table 4.7: Uses of the reflection method Class.getMethod, and the number and percentage that could be re-written using a structural downcast. Programs that did not call this method are omitted. The percentage entry in the last row is calculated by dividing the total "*could be rewritten*" by the total "*uses of getMethod.*"

sign that uses structural subtyping (Sect. 4.4). Additionally, some uses of Java reflection can be converted to uses of structural subtyping (Sect. 4.7). Together, this data supports hypothesis III.

Finally, the study in Sect. 4.6 showed the synergy between structural subtyping and external dispatch: hypothesis IV. The data showed that many cases of cascading instanceof tests in Java programs can improved if re-written using a combination of structural subtyping and external methods, a re-writing which allows an existing class to be adapted to a new context.

I hope that the results of this study will be used to inform designers of future programming languages, as well as serve as a starting point for further empirical studies in this area. Ultimately, one must study the way structural subtyping is eventually used by mainstream programmers; this work serves as a step in that direction.

Chapter 5

Additional Related Work

I have discussed the most closely related work in each appropriate chapter (see Sections 2.6, 2.4.2, 3.3, 3.8, and 4.3). In this chapter, I provide further detail and describe additional related work on the topics of structural subtyping, retroactive abstraction, external/multimethod dispatch, multiple inheritance, and related empirical studies.

5.1 Structural Subtyping

In Sect. 2.6, I showed that although languages with only nominal subtyping (such as Java) can be extended to provide support for multiple dispatch, these languages do not have the flexibility of languages with structural subtyping. In particular, such languages require programmers to declare (in advance) all the types they wish to use as abstractions, regardless of whether those types are required for dispatch.

Intersection types are useful for expressing combinations of types [Coppo and Dezani-Ciancaglini 1978; Coppo et al. 1979; Pottinger 1980; Büchi and Weck 1998], but they do not solve the problem of adding retroactive supertypes. Mixins and traits are designed mainly for code reuse [Bracha and Cook 1990; Ancona and Zucca 1996; Ducasse et al. 2006; Fisher and Reppy 2004], and therefore do not properly address the issues of retroactive abstraction and multimethods. I will revisit the topic of both mixins and traits in the context of the multiple inheritance features of Unity.

As previously mentioned, structural subtyping has been extensively studied in both formal and applied settings [Cardelli 1988; Bruce et al. 2003; Fisher and Reppy 1999; Leroy et al. 2004], but these formalisms and languages provide only internal dispatch. One of these languages, Moby, has particular similarities to Unity, as it both supports structural subtyping and a form of tag subtyping through its inheritance-based subtyping mechanism, this latter mechanism bearing similarity to sub-branding in Unity [Fisher and Reppy 1999; 2002]. This allows expressing many useful subtyping constraints, but Moby's class types are not integrated with object types in the same way as in Unity. For instance, in Moby, it is not possible to express the constraint that an object should have a particular class *and* should have some particular methods (that are not defined in the class itself). Additionally, the object-oriented core of Moby supports only internal dispatch. Moby does also include "tagtypes" that are very similar to brands in Unity.

These can be used to support downcasts or to encode multimethods, but they are disjoint from the object-oriented core of the system.

A more recent language design for structural subtyping is provided in Scala [Odersky and Zenger 2005; Odersky 2007], through type *refinements*. Since Scala also includes nominal types, intersection types allow combining the two components. However, Scala does not include general external dispatch, does not allow structural constraints to be placed on the receiver, and does not permit the definition of structural recursive types. Additionally, these aspects of Scala's type system have not yet been formalized (nor proved sound).

There has been a line of work that considers the question of how to add structural types to existing languages, such as C++ and Java [Baumgartner and Russo 1997; Laufer et al. 2000; Gil and Maman 2008]. The aforementioned work differs from Unity, however, in that it primarily focuses on implementation and integration issues.

As previously mentioned (Sect. 2.6), Cecil contains "where" clauses that can model some aspects of structural types, but they can only appear on top-level methods and can require verbose parameterization, in contrast to languages with true structural subtyping. Cecil has the most sophisticated form of "where" clause [Litvinov 1998; 2003; Chambers and the Cecil Group 2004], which originated in CLU [Liskov et al. 1977; Liskov 1983; Liskov and Wing 1993] and also appear in Theta [Liskov et al. 1994; Day et al. 1995] and PolyJ [Bank et al. 1997]. Of these languages, only Cecil supports external or multimethod dispatch.

Strongtalk presents a structural type system for Smalltalk and also supports named subtyping relationships through its "brand" mechanism [Bracha and Griswold 1993]. However, it is not possible to define subtyping on brands. Additionally, since it is a type system for Smalltalk, it supports only the single dispatch model.

The Modula-3 type system—from where I borrowed the term "brand"—has structural types with branding, but not structural sub*typing* [Nelson 1991]. That is, its type system will treat two record types as equivalent if they have the same structure but different type aliases, but does not recognize one as a subtype of the other if it has additional fields. The object-oriented part of the language solely uses nominal subtyping.

Other researchers have considered the problem of integrating nominal and structural subtyping. Reppy and Turon have addressed the problem in the context of typechecking traits [Reppy and Turon 2007]. Their resulting type system is a hybrid of nominal and structural subtyping. However, in their system, structural types are second-class; they apply to trait functions but not to expressions or ordinary functions. Consequently, there is less expressiveness as compared with Unity: it is not possible to constrain the argument of a function to have particular members, for example. Bono et al. have also proposed a type system that includes both nominal and structural aspects, but their system does not fully integrate the two disciplines [Bono et al. 2007]. The system only uses structural typing when typechecking uses of the this variable within a class, making their system considerably less expressive than Unity.

In the C++ concepts proposal, concepts can be either nominal or structural [Gregor et al. 2006]. However, concepts apply only to template constraints, not to the subtyping relation; in this way they are similar to the "where"-clause constraint-based polymorphism described above.

ML signature matching [Milner et al. 1997] is structural (rather than nominal), and as noted by Pierce and Harper [Pierce 2004], the design considerations of structural vs. nominal matching are very similar to those that arise when comparing structural vs. nominal subtyping.

Various calculi representing the ML module system distinguish between external labels and internal bound variables [Harper and Lillibridge 1994; Harper and Stone 2000]. This is somewhat analogous to the Unity distinction between simple and qualified names, though the motivation is different. In ML, internal bound variables may be α-renamed, while external labels may not. As this is purely a static concept, there is no notion of a runtime "mapping" from labels to variables.

5.2 Retroactive Abstraction

Supporting retroactive abstraction in a nominally-typed language is a troublesome issue, particularly its interaction with modular typechecking and compilation.

Global analyses. The approach of some languages is to give up on modularity entirely, and perform whole-program typechecking and compilation. This does provide some expressiveness advantages, but I believe these are far outweighed by the inherent scalability issues.

Examples of such languages are Cecil [Chambers and the Cecil Group 2004] and AspectJ [Kiczales et al. 2001]. Cecil has a clean separation of subtyping and inheritance and uses nominal subtyping. New subtyping relationships can be declared post-hoc, as can inheritance relationships (via "extension declarations"). This essentially adds new tags to objects after-the-fact. Such new inheritance relationships can affect the exhaustiveness and ambiguity checking of multimethods, a problem that is made tractable in Cecil due to its whole-program approach. Similarly, the whole-program analysis ensures that subtyping relationships (where there are no runtime tags involved) are consistent throughout the program.

AspectJ allows *inter-type declarations,* which allow new subtyping relationships (either extends or implements) to be added post-hoc [Kiczales et al. 2001]. This feature also allows adding methods to existing objects, which is similar in spirit to external methods. However, the added flexibility of these features comes at the cost of whole-program typechecking and compilation, the later which is achieved by "weaving" aspect declarations into the classes they extend.

Transitivity of subtyping. As modularity was an explicit design goal of this dissertation, I shall focus on work that attempts to reconcile modularity and retroactive abstraction. Unfortunately, it happens that there is a fundamental tension between retroactive (nominal) interface extension and modular compilation and typechecking.

First, to retroactively change a nominal subtype hierarchy, one must either abandon modular typechecking or change the subtype relation so that it is not transitive [Ostermann 2008]. To see why, suppose that we have definitions of interfaces I, J, and K, each with no declared relationship to the other. Now we add a retroactive declaration (outside of the definition of J) that $J <: I$, and another declaration (outside of the definition of K) that $K <: J$. With these declarations in place, the transitivity of subtyping gives us that $K <: I$. But, to establish this fact, the typechecker must know that 1) that it should use J as the intermediary type, 2) it should use the declaration $K <: J$, and 3) that it should use the declaration $J <: I$. But, the property to be proved, $K <: I$, does not mention J at all, nor is there any mention of J in the definitions of K and I. Since I may have any number of retroactive subtypes, this means that the typechecker must know of all subtypes of I—a global assumption. Put another way: to be modular, the typechecker cannot try to find the relationship $J <: I$, but at the same type it must be able to use $K <: J$ as the intermediary step in establishing that $K <: I$.

This problem does not arise in Java-like languages, precisely because the type hierarchy cannot be retroactively changed. To establish that $K <: I$, the typechecker simply follows the declared subtyping relations from K up to I. At no point does the typechecker need to produce an intermediary type that has not been explicitly declared as a supertype in the definition of some type along the way.

If retroactive abstraction is limited in some way, it can be compatible with modular typechecking. For example, Sather [Szyperski et al. 1993] allows a new interface to declare its implementing classes, but, for the sake of modular typechecking, this interface cannot be declared as extending some other interface [Stoutamire and Omohundro 1996].

The language $\mathrm{FJ}_{<:}$ addresses this problem by allowing flexible retroactive abstraction but changing the subtype relation so that it is not transitive [Ostermann 2008]. In particular, the programmer must manually provide a set of "witness" types so that the typechecker can apply subsumption. In this example, the programmer would have to first up-cast K to J then up-cast J to I so that the typechecker would see the desired relationship between K and I. As transitivity is very fundamental to subtyping, I believe this approach could be non-intuitive for programmers.

Tagged interfaces. The situation is exacerbated with tagged interfaces. Recall from Sect. 1.1 that I have defined an *interface* as a set of methods along with their types. Languages with nominal subtyping have *nominal* interfaces; these are analogous to Unity "type" abbreviations—with the additional restriction that a relationship between a class C and a nominal interface I must be explicitly stated in order for objects of class C to conform to type I. "Interfaces" with associated tags that can be used for dispatch (e.g., Java and C# interfaces) are termed "tagged interfaces."

Serious complications can arise when there is retroactive interface extension with tagged interfaces, as this allows programs to dispatch on the generated tags. In Java, for instance, programmers may perform "instanceof" tests, and Java extensions allow other forms of interface dispatch (e.g., predicate dispatch in JPred; multimethods in Nice [Bonniot 2007], Relaxed MultiJava [Millstein et al. 2003], and JavaGI [Wehr et al. 2007; Wehr and Thiemann 2009]). Of these, only JavaGI permits both retroactive interface extension and multiple dispatch, but this approach is inherently at odds with modular typechecking. In particular:

1. In the case where two nominal types have identical members, it would be useful to specify that the types should be considered equivalent (i.e., each is a subtype of the other). Unfortunately, this is quite problematic, as it would make the subtyping relation cyclic. This, in turn, causes problems for external/multimethod ambiguity checking.

2. It is unclear whether the retroactive extensions should have global or lexical scope. Neither approach is satisfactory. Suppose that class B implements I_1. In module M_1, we define a multimethod m that has a default case for class Object, a specialized case b_1 for I_1, and a specialized case b_2 for some other interface I_2. At this point, the call $m(\mathsf{new}\ B())$ would

execute b_1. Now, suppose that in a new module M_2, we add the declaration B implements I_2.

- With globally-scoped retroactive extension, we must either perform global typechecking, or allow some errors to be deferred to runtime (JavaGI takes the latter approach). If the new implements declaration has global scope, the call to m(new B()) is now ambiguous. Worse, in a new module M_3, a programmer could define I_2 extends I_1. Now the call is no longer ambiguous, but the behavior of the program has changed—b_2 is now executed!
- If retroactive extensions are locally scoped, this produces very odd subtyping semantics. In particular, what should happen if a module N that imports both C_1 and C_2 and calls m with a B object? Within C_1, $B \nleq I_2$, but within D, $B \leq I_2$. If the intended semantics is that b_2 should be executed, then within b_2, $B \leq_{\text{dynamic}} I_2$, but (statically), $B \nleq I_2$!

I believe the root of these problems is that here the *tag* hierarchy (on which dispatch can be performed) and the *type* hierarchy (used for subtyping) have been conflated. If the intended purpose of retroactive interface extension is to make a class compatible with a particular interface for the purposes of code reuse (e.g., to pass objects of the class to a library method), then it is unclear why there should be runtime tags associated with interfaces. On the other hand, if it is useful to allow dispatch on a hierarchy supporting multiple inheritance, one wonders (a) why these two concepts are not cleanly separated in these language designs and (b) if the added convenience of retroactive tag extension is worth the increase in complexity and reasoning ability.

To my knowledge, there is no type system with modular typechecking, a transitive nominal subtype relation, and retroactive (tagged/untagged) interface extension (where new nominal super- and sub-interfaces can be retroactively added). Unity sidesteps these issues with a clear separation between tag (i.e., brand) and type, and by disallowing dispatch on the structural component of a type.

5.3 External and Multimethod Dispatch

External and multimethod dispatch has been extensively studied, but in the context of either dynamically typed languages or languages with a purely nominal type system. Among the dynamically typed languages are Common Lisp [Steele, Jr. 1990; Paepcke 1993] and Dylan [Shalit 1997; Feinberg et al. 1997].

Some languages statically ensure that multimethod dispatch is exhaustive and unambiguous, but require that the entire program be available at compile-time. Examples of such languages include Cecil [Chambers 1992; Chambers and the Cecil Group 2004], Tuple [Leavens and Millstein 1998], and the Java extension Nice [Bonniot 2007].

More recent work has focused on modular typechecking of external methods and multimethods, as well as the problem of integrating external methods into existing languages; this includes the Dubious calculus (System M), MultiJava [Millstein and Chambers 2002; Clifton et al. 2006] and EML [Millstein et al. 2002; 2004]. I have built on these existing techniques for modular typechecking of external methods.

The language $\lambda\&$ [Castagna et al. 1995] includes multimethod dispatch and includes structural subtyping on methods, similar to Unity. However, the subtyping relation for object types (i.e., classes) uses only nominal subtyping, in contrast to Unity.

Case classes in Scala [Odersky 2007] can be used to obtain some of the benefits of external dispatch, but they are mainly useful for ML-style pattern matching. New functions can be added that perform case analysis on a case class hierarchy, but the signatures of the case classes do not change as with external methods. Scala case classes can either be *sealed*, in which case all cases must appear in the same compilation unit; otherwise they are extensible.

Finally, it is worth noting that a language supporting only external or multimethod dispatch (without structural subtyping or constraint-based polymorphism features) would lack the desired expressiveness. That is, retroactive abstraction of some form is necessary to solve the problems outlined in Chapter 1 (e.g., Fig. 1.7).

5.4 Multiple Inheritance

Here I describe related work that was not previously discussed in Sections 3.3 and 3.8.

Some uses of multiple inheritance can be encoded using *virtual classes* [Madsen and Moller-Pedersen 1989; Thorup 1997] or *nested inheritance* [Nystrom et al. 2004; 2006]. (For the purposes of this comparison, the differences between the two features are not particularly relevant; I use the newer "nested inheritance" terminology, however.) In languages with such features (e.g., Beta, Jx, J&), a class C may define one (or more) *nested classes* $\overline{D}$. When a new class E extends C, it inherits all of C's members, including the nested classes $\overline{D}$. As with ordinary inheritance, the meaning of C's code is different in the context of E, and E may override any of C's members—including the nested classes $\overline{D}$. When a nested class D_i is overridden in the new class E, the new definition *enhances*, rather than replaces, the old definition. Additionally, $E.D_i$ is a subclass (and, in nested inheritance, a subtype) of $C.D_i$.

As extensively detailed by Nystrom et al. [Nystrom et al. 2004; 2006], this latter feature provides powerful code reuse capabilities that are analogous to, but more powerful than, mixins. A particular strength of virtual classes and nested inheritance is the ability to reuse code *across* multiple related classes. Additionally, some uses of external methods can be encoded using these language features, with the interesting difference that the original classes (to which the external methods have been added) and the "new" classes induced by these additions can co-exist in the same program in the virtual class/nested inheritance encoding. Depending on the context, such a semantics could be useful. For these reasons, I view virtual classes and nested inheritance as features that are *complimentary* to my multiple inheritance design, despite the fact that these designs do aim to solve the common problem of code reuse). Additionally, to the best of my knowledge, there is no language that combines virtual classes/nested inheritance and external dispatch.

Cecil [Chambers 1992; Chambers and the Cecil Group 2004] also provides both multiple inheritance and multimethod dispatch, but it does not include constructors (and therefore provides ordinary dispatch semantics for methods acting as constructors), and it performs whole-program typechecking.

JavaGI allows multiple dispatch on interfaces, but performs class-load-time checking to detect any conflicts that may arise [Wehr et al. 2007; Wehr and Thiemann 2009].

Like JPred, the language Half & Half [Baumgartner et al. 2002] provides multimethod dispatch on Java interfaces. In this language, if there exist specialized method implementations for two incomparable interfaces *A* and *B*, the visibility of one of the two interfaces must be package-private. Like System M, this effectively disallows multiple (interface) inheritance across module boundaries (where a package is a module). Half & Half does not consider the problem of multiple inheritance with state.

Pirkelbauer et al have considered the problem of integrating multimethods into C++, which is especially difficult due to existing rules for overload resolution [Pirkelbauer et al. 2007]. However, this proposal is not modular; because of the potential for inheritance diamonds, the design requires link-time typechecking.

Note that it is possible to modify the semantics of multimethod dispatch so that, by definitions, ambiguities cannot arise in the presence of multiple inheritance. A language may linearize the class hierarchy (e.g., CLOS [Bobrow et al. 1988], [Agrawal et al. 1991]) or choose the appropriate method based on textual ordering [Boyland and Castagna 1997]. However, such semantics can be fragile and confusing for programmers.

5.5 Empirical Studies

As mentioned in Sect. 4.3, researchers have studied the problem of refactoring programs to use most general nominal types where possible [Forster 2006; Steimann 2007]. Structural subtyping would make such refactorings more feasible (since new types would not have to be defined) and applicable to more type references in the program (since structural supertypes for library types could be created, while new interfaces cannot).

Muschevici et al. measured the number of cascading instanceof tests in a number of Java programs, to determine how often multiple dispatch might be applicable [Muschevici et al. 2008]. They found that cascading instanceof tests were quite common, and that many cases could be rewritten to use multimethods; this is consistent with my results.

Corpus analysis is commonly used in empirical software engineering research. For example, it has been used to examine non-nullness [Chalin and James 2007], aspects [Baldi et al. 2008], micro-patterns [Gil and Maman 2005], and inheritance [Tempero et al. 2008].

Chapter 6

Conclusions

> There was a point to this story, but it has temporarily escaped the chronicler's mind.
>
> Douglas Adams (*The Hitchhiker's Guide to the Galaxy*)

The previous chapters have each provided evidence to support the hypotheses listed in Sect. 1.4. In this chapter, I describe in additional detail how each hypothesis has been validated, and how the hypotheses, taken together, support the main thesis of this dissertation. Additionally, I discuss potential limitations of the work as well as directions for future work.

6.1 Validation

Recall the thesis statement of Sect. 1.4:

> *An object-oriented programming language can provide integrated support for (a) external dispatch, (b) nominal subtyping, (c) structural subtyping, and (d) multiple inheritance—all without sacrificing modular typechecking. These richer structuring mechanisms can serve to make code more reusable and adaptable.*

The main thesis is supported by seven hypotheses, each of which is directly testable. There are two main types of hypothesis: those concerning modular typechecking and those concerning the language's expressiveness. The former type of hypothesis is supported through the Unity language design and proof of type safety and the latter type is supported through a combination of examples and empirical studies.

6.1.1 Modularity and Type Safety

As described in Chapters 2 and 3, the four main features of Unity, i.e, features (a)–(d), have interesting interactions with one another. For instance, as illustrated in Sect. 2.2.3, a naïve combination of structural subtyping and external dispatch would result in serious problems for ambiguity checking of external methods. Additionally, such a semantics could cause a program's behavior to be altered by the presence or absence of particular methods, which could in turn produce subtle bugs related to accidental subtyping.

Similarly, if typechecking is to be modular, combining external dispatch and multiple inheritance is non-trivial, the heart of the issue being the so-called "diamond problem." As described in Sect. 3.3, previous languages that include both of these features either restrict expressiveness or require that the programmer provide an exponential number of disambiguating methods. Neither of these solutions is satisfactory.

Consequently, two hypotheses directly address type safety and modular typechecking:

Hypothesis I. A language with synergy between structural subtyping and external dispatch can be achieved through a novel combination of structural and nominal subtyping.

Hypothesis V. Through the use of a novel multiple inheritance scheme, modular typechecking can be performed in a language with multiple inheritance and external dispatch, without requiring programmer-specified disambiguating methods.

The evidence for these two hypotheses is the full Unity language design (presented in Sections 2.5 and 3.7), the proof of type safety of the formal system (outlined in Sect. 3.7.4 and presented in Appendix A), and the detailed argument of modularity of the formal system (presented in Sect. 3.7.3).

6.1.2 Expressiveness

Designing a language is one matter, but designing a *useful* language is another matter entirely. Thus, five of the seven hypotheses concern the expressiveness and potential utility of the presented language design.

Hypothesis II. By providing retroactive abstraction, structural subtyping can be used to improve the reusability and maintainability of existing object-oriented programs.

This hypothesis is supported through the empirical studies described in Chapter 4. In particular, Sect. 4.3 presented evidence from a global analysis that inferred structural types for method parameters in Java programs. Quantitative analysis determined that many method parameters were overly precise, while qualitative analysis determined that, more often than not, it would be useful to generalize such method parameters. Since nominal subtyping is at odds with retroactively implementing/extending existing interfaces (described in Sect. 1.2 and in more detail in Sect. 5.2) *and* such a capability would be necessary when using types defined in libraries, it follows that structural subtyping would be beneficial in these situations. In particular, programmers could use structural types for method parameters, increasing their potential for reuse (as any object with a structurally-conforming implementation could passed in as the actual parameter). Additionally, the study showed that it is infeasible to define new (nominal) interfaces where possible; this strategy would result in 4.1 times as many interfaces in the resulting program, as compared to the original.

The study in Sect 4.5.1 found a high frequency of common methods (methods with the same name and signature, but that are not contained in a common supertype of the enclosing classes), and a low frequency of common methods that represent an accidental name clash (Sect 4.5.2). Having common methods between two classes may indicate a missed opportunity for abstraction, which could be solved using structural subtyping for retroactive abstraction. Finally, a complementary study found that common methods could result in code clones of a particular nature (Sect. 4.5.3).

Hypothesis III. Existing language designs can lead to coding patterns that defer errors to runtime; structural subtyping could provide more static typechecking in these situations by allowing programmers to encode more properties directly in the type system.

This hypothesis is again supported by empirical studies in Chapter 4. In particular, I presented quantitative and qualitative data showing that some Java runtime exceptions (i.e., OperationUnsupportedException) can be eliminated in a straightforward manner with a design that uses structural subtyping (Sect. 4.4). Additionally, another study found that some uses of Java reflection can be converted to uses of structural subtyping (Sect. 4.7).

Hypothesis IV. The combination of structural subtyping and external dispatch has the synergistic effect of providing an expressive form of retroactive abstraction.

This hypothesis is supported in two ways, the first being a series of illustrative examples in Sect. 2.2. In particular, the examples showed that structural subtyping allows defining abstract interfaces and external dispatch allows programmers to add new methods in order to make existing brands conform to those interfaces.

Next, the empirical study described in Sect. 4.6 showed that many cases of cascading instanceof tests in Java programs may be re-written using a combination of structural subtyping and external methods. Such a re-writing has many code reuse benefits, as it allows an existing class to be adapted to a new context.

Hypothesis VI. A language can be designed with a new form of multiple inheritance—multiple inheritance without diamonds—a design that provides more opportunities for code reuse and that is more expressive than other proposed alternatives to full multiple inheritance (i.e., multiple interface inheritance, mixins, and traits).

This hypothesis is validated by the Unity's multiple inheritance design (Chapter 3) as well as a detailed comparison to other language designs in the context of the Unity AST nodes example (Sect. 3.5). In particular, I showed that each of multiple interface inheritance, mixins, traits and Scala traits are less expressive than Unity, along the dimensions I identified. Multiple interface inheritance does not allow multiple inheritance of code, mixins do not allow one mixin to inherit from another, and (traditional) traits do not allow defining state.

Scala traits (which are a fusion of "ordinary" traits and mixins) are discussed in Sect. 3.3. The limitation of Scala traits is that their constructors may not take parameters (because of the potential for diamond inheritance), while Unity need not place such a restriction.[1] Additionally,

[1] Though the core calculus does not explicitly include constructors, their formalization is straightforward

if multiple dispatch or external dispatch were added to Scala, a strategy like that of JPred or Fortress would be necessary for ensuring unambiguity of multimethods/external methods.

Hypothesis VII. By converting inheritance diamonds to inheritance dependencies and subtyping among abstract classes (via a requires clause), a program with inheritance diamonds can be systematically translated into a program with multiple inheritance but without any inheritance diamonds.

This hypothesis is supported by the real-world examples in Sect. 3.6, which shows how C++ inheritance diamonds can be systematically translated to Unity. While I have presented examples from only two programs, I found at least 5 other (less interesting) inheritance diamonds in two other applications.

With regard to supporting this hypothesis, I believe a more systematic study of inheritance diamonds in C++ would *not* be especially worthwhile. Such a study could be interesting in its own right, but its value would mostly lie in characterizing the nature of C++ inheritance diamonds and quantifying how often concrete classes appear on the "sides" of the inheritance diamond (as the corresponding Unity translation would require converting this to an abstract class).

Instead, the real-world examples of Sect. 3.6 show that a) C++ programmers do make use of diamond inheritance, b) there exist such uses that are "reasonable," and (c) in some cases, the Unity translation is particularly trivial (when new concrete classes are not needed). The translation itself (outlined in Sect. 3.6.1) is so straightforward that I see little, if any, research contribution in its implementation.

6.2 Future Directions

The Unity design is not without its limitations. In particular, since a full implementation does not yet exist, it is quite conceivable that additional research problems could arise when implementing Unity as a general-purpose language. In this section, I outline several directions for future work.

6.2.1 Efficiency

It is unknown whether Unity code would be sufficiently efficient. This is tempered somewhat by the "pay-as-you-go" property of the language with regard to structural subtyping. That is, the name mapping dictionary is created only when structural types are explicitly needed. But, as I advocate a relatively high use of structural subtyping, the average execution overhead of method invocation may be quite high, in the absence of appropriate optimizations.

Of course, one should not underestimate the cleverness of those creating compilers, including "just-in-time" (JIT) compilers. The Whiteoak language, for example [Gil and Maman 2008] has achieved an acceptable level of performance by using a wrapper-generation mechanism for invoking structurally-typed methods. Additionally, Microsoft is actively supporting an effort to

and they are orthogonal to the issues under consideration.

integrate Ruby and Python into .NET, which is partly achieved by implementing a dynamic language runtime (DLR) on top of the regular common language runtime (CLR) [Microsoft 2009b]. The DLR, among other things, greatly eases the task of creating an efficient compiler for such dynamic languages, and includes functionality such as a shared dynamic type system and efficient generation of symbol tables [Chiles 2007]. C# 4.0 will also support a dynamic keyword that allows tagging some objects as dynamically typed [Microsoft 2009a]. On such objects, the typechecker will allow any method to be called and checking the existence of the method is deferred to runtime. While the merits of such a feature are debatable (and are mostly related to orthogonal issues such as importing dynamically linked libraries), its inclusion highlights the need for efficient dictionary-based method invocation.

In essence, I argue that it is quite conceivable that one could implement a Unity compiler that targets the DLR and achieves acceptable performance for method invocation. Implementing such a compiler, is, of course, relegated to future work.

Finally, it is quite possible that some applications will not have stringent performance demands. Quite a large amount of modern code is written in dynamic languages such as JavaScript, Ruby and Python, which are not known for their support of efficient method invocations.

6.2.2 Structural Downcasts

With the availability of structural subtyping, programmers may also wish to have structural downcasts. Such a feature can certainly be implemented, but it may be difficult to implement efficiently. It remains an open question whether structurally typed code would require a large number of structural downcasts, as well as how often the more efficient nominal downcast could be used instead.

6.2.3 Blame Assignment

As mentioned in Sect. 5.2, Ostermann has identified an important property of nominal subtyping: it allows a useful default *blame assignment* [Ostermann 2008]. In nominally typed languages, a class's author is the one responsible for ensuring that the class is adheres to the subtyping relationships induce by its implements clause. In contrast, with structural subtyping, it is always the client code's "fault" when a subtype relationship does not hold, and the error message will be noted at that point in the code. Ostermann argues that, instead, *flexible* blame assignment is key, so that programmers may specify who is responsible for maintaining a subtype relationship: either the user or the designer of a component. (Ostermann's language $FJ_{<:}$ allows such flexible blame assignment.)

As a concrete example, suppose it is intended that brand B always implement interface I. Type abbreviations in a Unity implementation would allow I to be defined in a single location, if desired, and used systematically throughout the program with the name I. Now, suppose a new method m is added to this single type abbreviation I, a method m that does not exist in B. In a nominally typed setting, the definition of B would be at fault, since it has declared an explicit relationship to I. In contrast, in a structurally typed setting, errors would appear whenever there was made an attempt to coerce an object with brand B to type I. In this example, the error message has appeared in the "wrong" location; this could certainly pose serious software engineering issues. (In fact, the designers of Strongtalk, which originally had both nominal and structural subtyping [Bracha and Griswold 1993], reverted to a purely nominal system in later versions, precisely due to blame assignment in error messages [Bracha 2006].)

However, I do not believe these problems are insurmountable. In particular, a combination of a methodology for type definitions and specialized IDE support could greatly minimize this problem. It is possible that one could move to a system that is a hybrid of Unity and Ostermann's $FJ_{<:}$, but the lack of transitive subtyping in that system is a worrisome issue.

6.2.4 Error messages

Related to the problem of blame assignment is that of producing readable error messages, which is generally much more difficult when a type system includes any structural aspects (which includes designs such as Java-like generics). The problem is that a particular structural type may have more than one corresponding user-defined type abbreviation, or—worse yet—it may have no such abbreviation. Such "anonymous" types can result in particularly indecipherable error messages, a problem of which (for example) users of C++ templates are painful aware.[2]

I believe this problem is an engineering issue at heart, as it is impossible to "prove" that a particular error message algorithm is "better" than others, let alone that it is "good." This is not to downplay the difficulty of solving the problem, but rather to identify what I believe would be a tractable approach. This problem is related to that of producing "good" type inference error messages (e.g., [Wand 1986; Beaven and Stansifer 1993; Duggan and Bent 1996; Tip and Dinesh 2001; Stuckey et al. 2003]), and it is possible that some of the approaches to the latter problem could inform a concrete solution to our error message problem.

6.2.5 IDE Support

The empirical study in Sect. 4.3 found that method parameters are often overly precise; I have argued that structural subtyping would make it easier to generalize these parameters. However, there is no reason to assume that users of a language with structural subtyping would use appropriately general types for parameters. Consequently, it could be quite useful to generalize the algorithm of the Infer Type refactoring [Steimann 2007] to apply in a structural setting. Such a refactoring would be even more powerful in a language with structural types, as the refactoring would be able to suggest types that are supertypes of existing library types (which the current refactoring cannot, as it applies to Java).

6.3 Broader Impact

The contributions of this thesis highlight the utility and importance of structural subtyping as a typing discipline. I have shown the synergy between structural subtyping and external dispatch,

[2] The new C++0x concepts proposal would improve the situation; interestingly, part of the solution involves introducing the notion of "nominal" vs. "structural" concepts [Gregor et al. 2006].

but I expect that an analogous synergy can be achieved when integrating structural subtyping and other language features.

In particular, the problem of accidental subtyping (discussed in Sections 2.2.2 and 4.5.2) would be greatly minimized if a richer type system were used. For example, if one were to use a lightweight form of pre- and post-conditions, such as typestates [Strom and Yemini 1986; Mandelbaum et al. 2003; DeLine and Fähndrich 2004; Bierhoff and Aldrich 2005; 2007; Kim et al. 2009], then similarly-named methods that perform different tasks can be differentiated. Subtyping between typestates, such as that proposed by [Bierhoff and Aldrich 2005; 2007], would be particularly useful in this setting. This idea could be extended even further to systems that check logical predicates, such as JML [Leavens et al. 2006] or Spec# [Barnett et al. 2004; 2005; Leino 2007]. In such a setting, subtyping would be simply become logical entailment; i.e., type $A \le B$ iff A implies B.

Another potential synergy could be harnessed when using co- and contravariant declaration-site type parameters along with structural subtyping.[3] Suppose co- and contravariant type parameters were prefixed with "+" or "-," respectively, such as in Scala [Odersky 2007]. Then, covariant (or contravariant) version of an interface would be a structural supertype of the invariant version of that interface. In particular, the following Scala code is valid:

```
trait ReadableCell [ +T ] {
    def get : T ;
}
trait WriteableCell [ -T ] {
    def set ( x : T ) : Unit ;
}

class Cell [ T ] ( init: T ) extends ReadableCell[T] with WriteableCell[T] {
    private var value: T = init
    def get: T = value
    def set ( x: T ): Unit = { value = x }
}
```

Here, the trait ReadableCell is a super-interface of Cell, and its type parameter T is *co*variant. Similarly, WriteableCell's type parameter is *contra*variant. Since invariant type parametrization is more restrictive than either co- or contravariant, Cell's type parameter is *in*variant.

Here, we have use nominal subtyping and set up explicit relationships between Cell, ReadableCell, and WriteableCell. With structural subtyping, however, we could retroactively add the types SReadableCell and SWriteableCell (making them structural types rather than traits) and Cell would be a subtype of each of these.

[3] Special thanks to Nick Benton for this observation.

Appendix A

Unity Type Safety

> The proof of the pudding is the eating.
>
> Miguel de Cervantes Saavedra (*Don Quixote*)

This section presents the full type safety proof that was outlined in Sect. 3.7.4. Definitions and lemmas that were originally mentioned in Chapter 3 appear here with new numbering (with backreferences to the previously stated versions).

A.1 Definitions

Note 1. The sequence $\overline{C}$ is shorthand for $\{C^{\,i\in 1..\#C}\}$, where $\#C$ is the length of the sequence $\overline{C}$.

Definition A.1 (Context consistency relation [Definition 3.3]).
The judgement $\Delta : \Sigma$ is defined by the following inference rules:

$$\text{(Delta-Wf-Empty)}\quad \frac{}{\cdot\ :\ \cdot}$$

$$\text{(Delta-Wf-Brand)}\quad \frac{\begin{array}{c}\Sigma = \Sigma_0, \textsf{brand}\ B(\sigma; \{q_i : B(M_i) \Rightarrow \tau_i{}^{\,i\in 1..n}\})\ \textsf{extends}\ \overline{C}\ \textsf{requires}\ \overline{D} \\ \Delta_0 : \Sigma_0 \qquad \textsf{fieldType}_\Sigma(\overline{D}) = \overline{\sigma} \\ \textsf{this} : B(M_i), \textsf{fields} : \sigma \wedge \overline{\sigma} \vdash_\Sigma e_i : \tau_i \quad (\forall\, i \in 1..n)\end{array}}{\Delta_0, B(q_i = e_i{}^{\,i\in 1..n})\ \textsf{extends}\ \overline{C}\ :\ \Sigma}$$

$$\text{(Delta-Wf-Method)}\quad \frac{\begin{array}{c}\Sigma = \Sigma_0, \textsf{method}\ q(q : B_i(M_i) \Rightarrow \tau_i{}^{\,i\in 1..n}) \qquad \Delta_0 : \Sigma_0 \\ \textsf{this} : B_i(M_i) \vdash_\Sigma e_i : \tau_i \quad (\forall\, i \in 1..n)\end{array}}{\Delta_0, \textsf{method}\ q(B_i.q = e_i{}^{\,i\in 1..n})\ :\ \Sigma}$$

Definition A.2 (Well-formed static context judgement).

The judgement Σ **ok** is defined by the following inference rules:

(Sigma-Wf-Base)

$$\frac{}{\cdot\ \mathbf{ok}}$$

(Sigma-Wf-Decl)

$$\frac{\textit{decl-type}_{\textit{name}} \notin \Sigma \qquad \Sigma\ \mathbf{ok} \qquad \Sigma \vdash \textit{decl-type}\ \mathbf{ok}}{\Sigma, \textit{decl-type}\ \mathbf{ok}}$$

Definition A.3 (Field type plus required field types).

$\mathrm{fieldWithReq}_{\Sigma}(B) \overset{\mathrm{def}}{=} \mathrm{fieldType}_{\Sigma}(B) \wedge \overline{\tau}_r$, where C requires $\overline{D} \in \Sigma$ and $\overline{\tau}_r = \mathrm{fieldType}_{\Sigma}(\overline{D})$.

A.2 Signature Weakening Lemmas

This section contains lemmas that prove weakening of the static context Σ for various judgements. These lemmas are used in both the progress and preservation proofs. Note that some auxiliary strengthening lemmas are needed; this is because the negation of some judgements are used (e.g., $\nvdash_{\Sigma}\ B.q$ **internal**).

Lemma A.1 (No duplicate names in well-formed contexts).

If Σ **ok** then all brand declarations $B \in \Sigma$ and all external method family declarations $q \in \Sigma$ are distinct from one another.

Proof. Straightforward induction on Σ **ok**. □

Lemma A.2 (Common method implies common ancestor [Lemma 3.3]).

If Σ **ok** and $\mathit{mtype}_{\Sigma}(q, B)$ and $\mathit{mtype}_{\Sigma}(q, C)$ then there exists some $D \neq \mathsf{Object}$ such that $B \sqsubseteq_{\Sigma} D$ and $C \sqsubseteq_{\Sigma} D$.

Proof. By simultaneous induction on the two mtype_{Σ} derivations, using the no-duplicate-names lemma (Lemma A.1), the constraints on owner brands (premises (1) and (3) of Tp-Ext-Method), and the fact that external methods cannot override internal methods (premise (4) of Tp-Ext-Method). □

Lemma A.3 (Weakening for sub-brand judgement).

If Σ_0 **ok** and $B \sqsubseteq_{\Sigma_0} C$ and Σ **ok** and $\Sigma \supseteq \Sigma_0$ then $B \sqsubseteq_{\Sigma} C$.

Proof. By induction on $B \sqsubseteq_{\Sigma_0} C$.

case Sub-Brand-Refl. Immediate.

case Sub-Brand-Trans. Straightforward induction.

case Sub-Brand-Decl. $B(\tau; Q)$ extends $C_1, \ldots, C_n \in \Sigma_0$

By the definition of the superset relation, $B \in \Sigma$. By the non-duplicate entries lemma (Lemma A.1), Σ has only one entry for B, therefore it is the same entry as that in Σ_0. □

Lemma A.4 (Weakening for subtype and sub-row judgements).

1. If Σ_0 **ok** and $\vdash_{\Sigma_0} t \leq t'$ and Σ **ok** and $\Sigma \supseteq \Sigma_0$, then $\vdash_{\Sigma} t \leq t'$.
2. If Σ_0 **ok** and $\vdash_{\Sigma_0} \overline{r_a : t_a} \trianglelefteq \overline{r_b : t_b}$ and Σ **ok** and $\Sigma \supseteq \Sigma_0$, then $\vdash_{\Sigma} \overline{r_a : t_a} \trianglelefteq \overline{r_b : t_b}$.

Proof. By mutual induction on the subtype and sub-row derivations.

1. By case analysis on the last rule used in the derivation $\vdash_{\Sigma_0} t \leq t'$:
   case Sub-Refl, Sub-$\wedge$L$_1$, Sub-$\wedge$L$_1$, Sub-Type-Var. Immediate.
   case Sub-Trans, Sub-Func, Sub-$\wedge$R, Sub-Mu. Straightforward, using induction hypothesis (1).
   case Sub-Record. Follows from induction hypothesis (2) and Sub-Record.
   case Sub-Obj. Follows from sub-brand weakening (Lemma A.3), induction hypothesis (2), and Sub-Obj.
   case Sub-Requires. Follows from non-duplicate brand definitions (Lemma A.1), the properties of set inclusion, induction hypothesis (2), and Sub-Requires.
   case Sub-Method. Follows from induction hypothesis (2), induction hypothesis (1), and Sub-Method.
2. By case analysis on the last rule used in the $\trianglelefteq$ derivation:
   case Sub-Row-Perm, Sub-Row-Width. Immediate.
   case Sub-Row-Depth. Follows from induction hypothesis (1) and Sub-Row-Depth.
   case Sub-Row-Trans. Straightforward, using induction hypothesis (2). □

Lemma A.5 (Strengthening for sub-brand judgement).
If Σ **ok** and $B \sqsubseteq_{\Sigma} C$ and Σ_0 **ok** and $\Sigma \supseteq \Sigma_0$ and $B \in \Sigma_0$, then $B \sqsubseteq_{\Sigma_0} C$.

Proof. By induction on $B \sqsubseteq_{\Sigma} C$.

case Sub-Brand-Refl. Immediate.

case Sub-Brand-Trans. Straightforward induction.

case Sub-Brand-Decl. brand B extends $C \in \Sigma$.
By the non-duplicate entries lemma (Lemma A.1), Σ_0 has only one entry for B, therefore brand B extends $C \in \Sigma_0$. The result follows from Sub-Brand-Refl. □

Lemma A.6 (Weakening for *inherit-ok* judgement).
If Σ_0 **ok** and $\vdash_{\Sigma_0} B$ extends $\overline{C}$ requires $\overline{D}$ **inherit-ok** and Σ **ok** and $\Sigma \supseteq \Sigma_0$ and $B \notin \Sigma$, then $\vdash_{\Sigma} B$ extends $\overline{C}$ requires $\overline{D}$ **inherit-ok**.

Proof. By inversion on $\vdash_{\Sigma_0} B \cdots$ **inherit-ok**. We proceed by considering each premise of **inherit-ok** and showing that the same premise holds under Σ.

1. $\overline{C} \in \Sigma_0$. By the definition of set inclusion, $\overline{C} \in \Sigma$.

2. $\overline{D} \in \Sigma_0$. Similar to above.
3. $D_i \notin \overline{C}$ $(\forall i \in 1..m)$. Immediate.
4. $\forall i, j.\, i \neq j. \nexists D'. C_i \sqsubseteq_{\Sigma_0} D'$ and $C_j \sqsubseteq_{\Sigma_0} D'$, where $D' \neq \mathsf{Object}$.
   Suppose that for some $i \neq j$, there exists $D' \neq \mathsf{Object}$ such that $C_i \sqsubseteq_{\Sigma} D'$ and $C_j \sqsubseteq_{\Sigma} D'$. But, by sub-brand strengthening (Lemma A.5), this implies that $C_i \sqsubseteq_{\Sigma_0} D'$ and $C_j \sqsubseteq_{\Sigma_0} D'$, which is a contradiction of our initial assumption. Therefore, the no-diamond property holds under Σ as well as Σ_0.
5. C_i $\mathsf{requires}$ $E \in \Sigma_0$ implies $\exists k.\ C_k \sqsubseteq_{\Sigma_0} E$ or $D_k \sqsubseteq_{\Sigma_0} E$ $(\forall i \in 1..n)$.
   Since $\overline{C} \in \Sigma$ and Σ does not contain duplicate brands (Lemma A.1), the declarations of $\overline{C}$ in Σ are the same as those in Σ_0. Next, by sub-brand weakening (Lemma A.3), if $C_k \sqsubseteq_{\Sigma_0} E$, then $C_k \sqsubseteq_{\Sigma} E$, and similarly for D_k.
6. D_i $\mathsf{requires}$ $E' \in \Sigma_0$ implies $\exists k.\ C_k \sqsubseteq_{\Sigma_0} E'$ or $D_k \sqsubseteq_{\Sigma_0} E'$ $(\forall i \in 1..m)$. Similar to above.
7. $\forall i, j. \forall q.\ \mathit{mtype}_{\Sigma_0}(q, C_i) = \rho$ and $\mathit{mtype}_{\Sigma_0}(q, C_j) = \rho'$ implies $i = j$.
   From this it follows that for all $i \neq j$, $\nexists \mathit{mtype}_{\Sigma_0}(q, C_i)$ or $\nexists \mathit{mtype}_{\Sigma_0}(q, C_j)$. We must show that the above holds under the larger context Σ. Suppose that for some i and j, $i \neq j$ and $\mathit{mtype}_{\Sigma}(q, C_i) = \rho$ and $\mathit{mtype}_{\Sigma}(q, C_j) = \rho'$. By Lemma A.2, there exists D such $D \neq \mathsf{Object}$ and $C_i \sqsubseteq_{\Sigma} D$ and $C_j \sqsubseteq_{\Sigma} D$. But, this is a contradiction of the no-diamond property shown above (item 4). Therefore, the required property holds under Σ as well as Σ_0. □

Lemma A.7 (Weakening for *internal* judgement).
If Σ_0 **ok** and $\vdash_{\Sigma_0} B.q$ **internal** and Σ **ok** and $\Sigma \supseteq \Sigma_0$ and, then $\vdash_{\Sigma} B.q$ **internal**.

Proof. Straightforward induction on $\vdash_{\Sigma_0} B.q$ **internal**. □

Lemma A.8 (Strengthening for *internal* judgement).
If Σ **ok** and $\vdash_{\Sigma} B.q$ **internal** and Σ_0 **ok** and $\Sigma \supseteq \Sigma_0$ and $B \in \Sigma_0$, then $\vdash_{\Sigma_0} B.q$ **internal**

Proof. By induction $\vdash_{\Sigma} B.q$ **internal**.

case Internal-Base. Follows from the fact that $B \in \Sigma_0$ and that there is only one entry for B in Σ (Lemma A.1).

case Internal-Inh. From the fact that $B \in \Sigma_0$ and that there is only one entry for B in Σ (Lemma A.1), it follows that $\mathsf{brand}\ B(\tau; Q)\ \mathsf{extends}\ \overline{C} \in \Sigma_0$ and $q \notin Q$. Since super-brands of B are also in Σ (Lemma A.12), $\overline{C} \in \Sigma_0$. The result then follows from the induction hypothesis and Internal-Inh. □

Lemma A.9. If Σ **ok** and $\mathit{mtype}_{\Sigma}(q, B) = \rho$ and $\mathsf{method}\ C.q(\cdots) \notin \Sigma$, then $\vdash_{\Sigma} B.q$ **internal**.

Proof. Straightforward induction on the *mtype* derivation, using the fact that MType-Ext cannot apply. □

Lemma A.10 (Weakening for *mtype* judgement).
If Σ_0 **ok** and $\mathit{mtype}_{\Sigma_0}(q, B) = \rho$ and Σ **ok** and $\Sigma \supseteq \Sigma_0$, then $\mathit{mtype}_{\Sigma}(q, B) = \rho$.

Proof. By induction on $\mathit{mtype}_{\Sigma_0}(q, B) = \rho$.

case MType-Base. By the definition of set inclusion and the fact that well-formed contexts do not contain duplicate brands (Lemma A.1), brand $B(\tau; \ldots, q : \rho, \ldots) \in \Sigma$. The result then follows from MType-Base.

case MType-Ext. Similar to above.

case MType-Inh.
(1) brand $B(\tau; Q)$ extends $\overline{C} \in \Sigma_0$
(2) $q \notin Q$
(3) $\nexists \mathit{extDef}_{\Sigma_0}(q, B)$
(4) $\exists k.\, \mathit{mtype}_{\Sigma_0}(q, C_k) = \rho$

By the definition of set inclusion and the property of no duplicate brands in a well-formed context (Lemma A.1), the first two premises hold for Σ. It suffices to show that $\nexists \mathit{extDef}_{\Sigma}(q, B)$. Then, by the induction hypothesis, $\mathit{mtype}_{\Sigma}(q, C_k) = \rho$ and the result follows from MType-Inh.

We are to show $\nexists \mathit{extDef}_{\Sigma}(q, B)$. Either (a) an external method family q exists in Σ_0 or (b) it does not:

subcase (a) method $B.q(\ldots) \in \Sigma_0$.
We have method $q \in \Sigma$ (by the properties of set inclusion). Since we have $\nexists \mathit{extDef}_{\Sigma_0}(q, B)$ and methods must be defined in a single block, this yields $\nexists \mathit{extDef}_{\Sigma}(q, B)$.

subcase (b) method $q \notin \Sigma_0$.
By Lemma A.9, $\vdash_{\Sigma_0} C_k$ **internal**. By Internal-Inh, $\vdash_{\Sigma_0} B.q$ **internal**. By Lemma A.7, this implies (B) $\vdash_{\Sigma} B.q$ **internal**.
Again, there are two cases: either (i) method $q \notin \Sigma$ or (ii) method $q \in \Sigma$:

(i). Result follows from the definition of *extDef*.

(ii). Let *decl-type* = method $D.q(B.q : \rho, \ldots)$ and let $\Sigma = \Sigma_1, \mathit{decl\text{-}type}, \Sigma_2$. Inversion on $\Sigma_1 \vdash \mathit{decl\text{-}type}$ **ok** yields premise (4) of Tp-Ext-Method: $\nvdash_{\Sigma_1} B.q$ **internal**. Since $\Sigma \supseteq \Sigma_1$, we may use the contrapositive of strengthening for the **internal** judgement (Lemma A.8), which yields $\nvdash_{\Sigma} B.q$ **internal**. This is a contradiction of (B), therefore, this case is vacuous. □

Lemma A.11 (Weakening for *override-ok* judgement).
If Σ_0 **ok** and $\vdash_{\Sigma_0} B.q : \rho$ **override-ok** and Σ **ok** and $\Sigma \supseteq \Sigma_0$, then $\vdash_{\Sigma} B.q : \rho$ **override-ok**.

Proof. Follows from *mtype* weakening (Lemma A.10) and subtype weakening (Lemma A.4). □

Lemma A.12 (Super-brands exist in well-formed contexts).
If $B \sqsubseteq_\Sigma C$ and Σ **ok** then $C \in \Sigma$.

Proof. Straightforward induction on Σ **ok**, using inversion of *decl-type* **ok** and premises (1) and (2) of **inherit-ok** in the base case. □

Lemma A.13 (Weakening of *decl-type ok* judgement).

If Σ_0 **ok** and $\Sigma_0 \vdash$ *decl-type* **ok** and
Σ **ok** and $\Sigma \supseteq \Sigma_0$ and
$\mathit{decl\text{-}type}_{name} \notin \Sigma$,
then $\Sigma \vdash$ *decl-type* **ok**.

Proof. By inversion on $\Sigma_0 \vdash$ *decl-type* **ok**.

case Tp-Brand-Decl. We proceed by examining each premise and showing that it holds under Σ; then, rule Tp-Brand-Decl can be used to derive $\Sigma \vdash$ *decl-type* **ok**.

1. $\vdash_{\Sigma_0} B$ extends $\overline{C}$ requires $\overline{D}$ **inherit-ok**.
   Follows from weakening on **inherit-ok** judgement (Lemma A.6).
2. $\Sigma'_0 = \Sigma_0, \mathit{decl\text{-}type}$.
   Take $\Sigma' = \Sigma, \mathit{decl\text{-}type}$. We have Σ' **ok** since by assumption, Σ_0 **ok** and $\Sigma_0 \vdash$ *decl-type* **ok**; Sigma-Wf-Decl then applies.
3. $\vdash_{\Sigma'_0} \tau \leq \mathrm{fieldType}_{\Sigma_0}(\overline{C})$.
   Follows from weakening on subtype judgement (Lemma A.4).
4. $\overline{q}$ distinct. Immediate.
5. $\vdash_{\Sigma'_0} B.\overline{q : \rho}$ **override-ok**.
   Follows from weakening on the **override-ok** judgement (Lemma A.11).

case Tp-Ext-Method. We proced as in the previous case, showing that each premise holds under Σ.

1. $B \neq \mathsf{Object}$. Immediate.
2. $\overline{C}$ distinct. Immediate.
3. $C_i \sqsubseteq_{\Sigma_0} B \ \ (\forall\, i \in 1..n)$.
   Follows from sub-brand weakening (Lemma A.3).
4. $\nvdash_{\Sigma_0} \ \overline{C}.q$ **internal**. Follows from the contrapositive form of strengthening for **internal** (Lemma A.8).
5. $\Sigma'_0 = \Sigma_0, \mathit{decl\text{-}type}$. Take $\Sigma' = \Sigma_0, \mathit{decl\text{-}type}$.
6. $\vdash_{\Sigma'_0} \overline{C}.q : \overline{\rho}$ **override-ok**.
   Follows from weakening on the **override-ok** judgement (Lemma A.11). □

Lemma A.14 (Declarations are well-typed under their containing context).
If Σ **ok** and *decl-type* $\in \Sigma$ then $\Sigma \vdash$ *decl-type* **ok**.

Proof. Straighforward induction on Σ **ok**, using Lemma A.13 for the base case. □

Lemma A.15 (Weakening for typing judgement).
If Σ_0 **ok** and $\Gamma \vdash_{\Sigma_0} e : \tau$ and Σ **ok** and $\Sigma \supseteq \Sigma_0$, then $\Gamma \vdash_\Sigma e : \tau$.

Proof. By induction on $\vdash_{\Sigma_0} e : \tau$.

case Tp-Var, Tp-Unit. Immediate.

case Tp-Fun, Tp-App, Tp-New-Record, Tp-Proj, Tp-Fold, Tp-Unfold. Straightforward induction.

case Tp-Subs. Follows from weakening on the subtype relation (Lemma A.4).

case Tp-New-Obj. By the fact that declarations are distinct from one another (Lemma A.1) and the properties of set inclusion, B requires $\bullet$ and fieldType$_\Sigma(B) = \tau$. By the induction hypothesis, $\Gamma \vdash_\Sigma e_1 : \tau$. By weakening on the *mtype* judgement, $mtype_\Sigma(\overline{q}, B) = \overline{\rho}$. The result then follows from Tp-New-Obj.

case Tp-With. Similar to above.

case Tp-Invoke-Struct. Follows from the induction hypothesis on the first premise, weakening of the subrow judgement (Lemma A.4), and Tp-Invoke-Struct.

case Tp-Invoke-Nom. Follows from the induction hypothesis on the first premise, weakening of the *mtype* judgement (Lemma A.10), weakening of subrow judgement (Lemma A.4) and Tp-Invoke-Nom.

case Tp-Invoke-Super. Similar to above. □

A.3 Basic Lemmas

This section contains lemmas used by both progress and preservation proofs.

A.3.1 Miscellaneous Lemmas

Lemma A.16 (Consistency of static and dynamic sub-branding).
If Σ **ok** and $\Delta : \Sigma$ then $B_1 \sqsubseteq_\Sigma B_2$ iff $B_1 \sqsubseteq_\Delta B_2$.

Proof. For the "if" direction, straightforward induction on $B_1 \sqsubseteq_\Sigma B_2$. For the "only if" direction, straightforward induction on $B_1 \sqsubseteq_\Delta B_2$, in both cases, using the definition of $\Delta : \Sigma$ (Def. 3.2). □

Lemma A.17 (Precisifying object types).
If Σ **ok** and $\Gamma \vdash_\Sigma \widehat{C}(e; \overline{n \hookrightarrow q}) : \tau$ and $\vdash_\Sigma \tau \leq B(M)$, then

1. $\Gamma \vdash_\Sigma \widehat{C}(e; \overline{n \hookrightarrow q}) : C(\overline{n : \rho})$, where $mtype_\Sigma(\overline{q}, C) = \overline{\rho}$ and $\vdash_\Sigma C(\overline{n : \rho}) \leq B(M)$, and

2. $\overline{n}$ distinct, and
3. C requires $\bullet \in \Sigma$.

Proof. By induction on the typing derivation.

case Tp-Subs. Follows from the induction hypothesis and Sub-Trans.
case Tp-Obj. Immediate from premises of rule. □

Lemma A.18 (Reflexivity of sub-row judgement).
The sub-row judgement is reflexive; i.e., $\Gamma \vdash_\Sigma \overline{r : t} \trianglelefteq \overline{r : t}$, for all Σ such that Σ **ok**.

Proof. Immediate from either Sub-Row-Perm or Sub-Row-Depth. □

Lemma A.19 (Reflexivity and transitivity of method subtyping).
The following rules are admissible:

$$\frac{}{\rho \leq \rho} \qquad\qquad \frac{\rho_1 \leq \rho_2 \qquad \rho_2 \leq \rho_3}{\rho_1 \leq \rho_3}$$

Proof. For the reflexivity property, result follows from Sub-Brand-Refl and reflexivity of sub-row (Lemma A.18). Transitivity proof is straightforward. □

Lemma A.20 (Properties of valid overrides).
If Σ **ok** and $\vdash_\Sigma B.q : \rho'$ **override-ok** and $B \sqsubseteq_\Sigma C$ and $mtype_\Sigma(q, C) = \rho$, then $\rho' \leq \rho$.

Proof. Straightforward induction on $B \sqsubseteq_\Sigma C$, using inversion of the **override-ok** derivation in the base case. □

Lemma A.21 (Sub-brands have *mtype*s that are subtypes).
If Σ **ok** and $B \sqsubseteq_\Sigma C$ and $mtype_\Sigma(q, C) = \rho$, then $mtype_\Sigma(q, B) = \rho'$, where $\vdash_\Sigma \rho' \leq \rho$.

Proof. By induction on $B \sqsubseteq C$.

case Sub-Brand-Refl. Result follows from reflexivity of method subtyping (Lemma A.19).

case Sub-Brand-Trans. Straightforward uses of induction hypothesis, followed by transitivity of method subtyping (Lemma A.19).

case Sub-Brand-Decl. B extends C
There are four possible cases:

1. q is defined internally in B with type ρ'. By MType-Base, $mtype_\Sigma(q, B) = \rho'$. Since Σ is well-formed, by Lemma A.14, the declaration of B is well-formed under Σ. By inversion on *decl-type* **ok**, $B.q : \rho'$ **override-ok** (premise (5) of Tp-Brand-Decl). By the properties of a valid override (Lemma A.20), $\rho' \leq \rho$, which is the required result.

2. q is defined externally for B with type ρ'. Similar to above.
3. q is not defined in B. By MType-Inh, $\mathit{mtype}_\Sigma(q, B) = \rho$, which is the required result. □

Lemma A.22 (*lookup* implies *mtype*).
If Σ **ok** and $\Delta : \Sigma$ and $\mathit{lookup}_\Delta(q, B) = e$, then there exists a derivation $\mathit{mtype}_\Sigma(q, B) = \rho$, for some ρ.

Proof. By induction on the derivation of lookup_Δ.

case Lookup-Base. Follows from the definition of $\Delta : \Sigma$ and MType-Base.

case Lookup-Ext. Follows from the definition of $\Delta : \Sigma$ and MType-Ext.

case Lookup-Inh. We have $\mathit{lookup}_\Delta(q, C_k) = e$, where B extends $C_k \in \Delta$. Also, by $\Delta : \Sigma$, we have $C_k \in \Sigma$. By the induction hypothesis, $\mathit{mtype}_\Sigma(q, C_k) = \rho$. Applying MType-Inh yields the required result. □

Lemma A.23 (Properties of *super*).
If Σ **ok** and $\Delta : \Sigma$ and $\mathit{super}_\Delta(B \text{ as } C) = D$ then $B \sqsubseteq_\Delta D$ and $D \sqsubseteq_\Delta C$.

Proof. By induction on $\mathit{super}_\Delta(B \text{ as } C) = D$.

case Super-Base. By Sub-Brand-Decl, $B \sqsubseteq_\Delta D$. $D \sqsubseteq_\Delta C$ by premise of rule.

case Super-Inh. By the induction hypothesis, $E_k \sqsubseteq_\Delta D$ (where B extends $E_k \in \Delta$) and $D \sqsubseteq_\Delta C$. By Sub-Brand-Decl and Sub-Brand-Trans, $B \sqsubseteq_\Delta D$, which is the required result. □

Lemma A.24 (No diamond inheritance).
If Σ **ok** and brand B extends $\overline{C} \in \Sigma$, then there does not exist D (other than Object), such that $C_i \sqsubseteq_\Sigma D$ and $C_j \sqsubseteq_\Sigma D$ (for any $i \neq j$).

Proof. By Lemma A.14, $\Sigma \vdash$ brand $B \cdots$ **ok**. By inversion on the *decl-type* **ok** judgement, we have $\vdash_\Sigma$ brand $B \cdots$ **inherit-ok** (premise (1) of Tp-Brand-Decl). Inverting this last judgement yields premise (4) of Tp-Inherit, which is the required result. □

Lemma A.25 (Well-formed brands do not contain duplicate methods).
If Σ **ok** and brand $B(\tau; \overline{q : \rho}) \cdots \in \Sigma$, then there are no duplicates in $\overline{q}$.

Proof. Follows from Lemma A.14 and premise (4) of Tp-Brand-Decl. □

Lemma A.26 (Properties of well-formed external methods).
If Σ **ok** and method $B.q(\overline{C}.q : \overline{\rho}) \in \Sigma$, then we have all of the following:

1. $B \neq$ Object
2. $\overline{C}$ distinct

3. $C_i \sqsubseteq_\Sigma B \ (\forall i \in 1..\#C)$
4. $\nvdash_\Sigma \overline{C}.q$ **internal**.

Proof. Straightforward induction on Σ **ok**, using the fact that $\vdash_\Sigma$ $\mathsf{method}\ \ldots$ **ok** (by Lemma A.36) and then applying inversion on this derivation. □

A.3.2 Inversion Lemmas

Lemma A.27 (Inversion of the sub-row judgement).
If Σ **ok** and $\Gamma \vdash_\Sigma \{r_{a_i} : t_{a_i}\ ^{i\in 1..n}\} \trianglelefteq \{r_{b_j} : t_{b_j}\ ^{j\in 1..m}\}$, then $\{r_{b_j}\ ^{j\in 1..m}\} \subseteq \{r_{a_i}\ ^{i\in 1..n}\}$ ($\overline{r}_a$ includes at least the labels in $\overline{r}_b$) and $\Gamma \vdash_\Sigma t_{a_i} \le t_{b_j}$ for each common label $r_{a_i} = r_{b_j}$.

Proof. Straightforward induction on the $\trianglelefteq$ derivation. □

Lemma A.28 (Inversion of subtyping [expression and method types]).

1. If Σ **ok** and $\Gamma \vdash_\Sigma \tau_1 \to \tau_2 \le \sigma_1 \to \sigma_2$, then $\Gamma \vdash_\Sigma \sigma_1 \le \tau_1$ and $\Gamma \vdash_\Sigma \tau_2 \le \sigma_2$.
2. If Σ **ok** and $\Gamma \vdash_\Sigma \{\overline{\ell : \tau}\} \le \{\overline{k : \sigma}\}$, then $\Gamma \vdash_\Sigma \{\overline{\ell : \tau}\} \trianglelefteq \{\overline{k : \sigma}\}$
3. If Σ **ok** and $\Gamma \vdash_\Sigma \mu X.\tau_1 \le \mu Y.\tau_2$, then $\Gamma, X \le Y \vdash \tau_1 \le \tau_2$.
4. If Σ **ok** and $\Gamma \vdash_\Sigma B_1(M_1) \le B_2(M_2)$, then $\Gamma \vdash_\Sigma M_1 \trianglelefteq M_2$ and either (a) $B_1 \sqsubseteq_\Sigma B_2$ or (b) there exists some B_2' where $B_2' \sqsubseteq_\Sigma B_2$ and B_1 $\mathsf{requires}$ $B_2' \in \Sigma$.
5. If Σ **ok** and $\Gamma \vdash_\Sigma M_1 \Rightarrow \tau_1 \le M_2 \Rightarrow \tau_2$ then $\Gamma \vdash_\Sigma M_2 \trianglelefteq M_1$ and $\Gamma \vdash_\Sigma \tau_1 \le \tau_2$.

Proof. Straightforward induction on each subtyping derivation. □

Lemma A.29 (Inversion of the typing judgement).

1. If $\Gamma \vdash \lambda x{:}\tau_1.\, e : \sigma$ and $\sigma \le \sigma_1 \to \sigma_2$ then $\sigma_1 \le \tau_1$ and $\Gamma, x : \tau_1 \vdash e : \sigma_2$.
2. If $\Gamma \vdash (\ell = e\ ^{i\in 1..n}) : \sigma$ and $\sigma \le \{k : \tau\ ^{j\in 1..m}\}$, then $e_i : \tau_j$ for each common label $\ell_i = k_j$.
3. If $\Gamma \vdash \mathsf{fold}_{\mu X.\tau}\, e : \sigma$ and $\sigma \le \mu X.\tau$, then $\Gamma \vdash e : [\mu X.\tau / X]\,\tau$.
4. If $\Gamma \vdash \widehat{C}(e; \overline{n_a \hookrightarrow q_a}) : \sigma$ and $\sigma \le B(\overline{n_b : \rho_b})$ then:
   (a) $C \sqsubseteq B$;
   (b) $\Gamma \vdash e : \text{fieldType}_\Sigma(C)$; and
   (c) $\overline{n_a : \rho_a} \trianglelefteq \overline{n_b : \rho_b}$, where $mtype_\Sigma(\overline{q}_a) = \overline{\rho}_a$.

Proof. By induction on each typing derivation. Note that for each case, the derivation ends in exactly one of two rules, one of which is always Tp-Subs.

1. Functions. Straightforward, using the induction hypothesis for case Tp-Subs and the subtyping inversion lemma (Lemma A.28) for case Tp-Fun.
2. Records. Straightforward induction, using the subtyping inversion lemma (Lemma A.28) for the base case.

3. Recursive types. Straightforward induction, using Lemma A.28 for the base case.
4. Object types. $\widehat{C}(e; n_i \hookrightarrow q_i{}^{i\in 1..n}) : \sigma$

   case Tp-Subs. Result follows from the induction hypothesis and Sub-Trans.

   case Tp-New-Obj. Follows from subtype inversion, sub-row inversion, and the rule's premise that C requires $\bullet \in \Sigma$. □

A.4 Progress Lemmas and Theorem

Lemma A.30 (Canonical forms).
Suppose Σ **ok** and $\vdash_\Sigma v : \sigma$ and $\vdash_\Sigma \sigma \leq \tau$.

1. If $\tau = \mathsf{unit}$ then $v = ()$.
2. If $\tau = \tau_1 \to \tau_2$ then v is of the form $\lambda x{:}\tau_1'.\, e$.
3. If $\tau = B(\overline{n : \rho})$ then v is of the form $\widehat{C}(v'; \overline{n_a \hookrightarrow q_a})$.
4. If $\tau = \{\overline{\ell : \tau}\}$ then v is of the form $(\overline{k = v})$.
5. If $\tau = \mu X.\tau$ then v is of the form $\mathsf{fold}_\sigma\ v'$.

Proof. Straightforward induction on typing derivations. □

Lemma A.31 (*lookup* defined on well-typed methods [Lemma 3.4]).
If Σ **ok** and $\mathit{mtype}_\Sigma(q, B) = \rho$ and $\Delta : \Sigma$, then $\mathit{lookup}_\Delta(q, B) = e$, for some unique e.

Proof. By induction on $\mathit{mtype}_\Sigma(q, B) = \rho$.

case MType-Base. q is defined directly in B.
From the definition of $\Delta : \Sigma$, the rule Lookup-Base applies. Since brands do not contain duplicate methods (Lemma A.25), there is only one applicable $q = e$ for B. It now suffices to show that the rule Lookup-Ext cannot apply, as Lookup-Inh is already excluded (its second premise does not hold).

Suppose Lookup-Ext did apply. By the definition of $\Delta : \Sigma$, there is a definition *decl-type* = method $D.q(\ldots, B.q : \rho', \ldots) \in \Sigma$. By Lemma A.26, $\nvdash_\Sigma B.q$ **internal**. But, by assumption, q is defined in B and therefore rule Internal-Base applies, yielding $\vdash_\Sigma B.q$ **internal**—a contradiction.

case MType-Ext. From the definition of $\Delta : \Sigma$, the rule Lookup-Ext applies. Since there are no duplicate method families q in Σ (Lemma A.1), inversion on $\Delta : \Sigma$ yields that is only one such entry method $q \cdots \in \Delta$. Let *decl-type* = $D.q(\overline{C}.q : \overline{\rho})$ be the definition of the method family q in Σ. By Lemma A.26, $\overline{C}$ distinct. By inversion on $\Delta : \Sigma$, there is not a duplicate entry $B.q = e'$ in the method family q in Δ. Therefore, there is exactly one derivation of Lookup-Ext.

It now suffices to show that Lookup-Base does not apply, as by inspection, Lookup-Inh does not apply. The reasoning for this is similar to that above; Lemma A.26 yields $\nvdash_{\Sigma} B.q$ **internal**, from which it follows that Lookup-Base does not apply.

case MType-Inh. We have $\mathit{mtype}_{\Sigma}(q, C_k) = \rho$, for some C_k where $B \;\mathsf{extends}\; C_k$. By the induction hypothesis, $\mathit{lookup}_{\Delta}(q, C_k) = e$, for a unique e. It now suffices to show that there does *not* exist $j \neq k$ such that $B \;\mathsf{extends}\; C_j$ and $\mathit{lookup}_{\Delta}(q, C_j) = e'$. Then, the rule Lookup-Inh then uniquely applies, since its premises exclude the other two rules.

Suppose such a j did exist, i.e., $B \;\mathsf{extends}\; C_j$ and $\mathit{lookup}_{\Delta}(q, C_j) = e'$. By Lemma A.22 (*lookup* implies *mtype*), there exists ρ' where $\mathit{mtype}_{\Sigma}(q, C_j) = \rho'$.

By Lemma A.36, $\Sigma \vdash (B(\ldots) \;\mathsf{extends}\; C_j, C_k, \ldots \mathsf{requires}\; \ldots)$ **ok**. Inversion on *decl-type* **ok** yields $B \;\mathsf{extends}\; C_j, C_k \ldots$ **inherit-ok**; i.e., premise (1) of Tp-Brand-Decl. Inversion on this last derivation yields premise (7) of Tp-Inherit, which assumes that $j \neq k$. As this is a contradiction of the assumption above, this yields the required result. □

Lemma A.32 (Well-typed objects have well-typed simple names).
If $\Gamma \vdash \widehat{C}(v; \overline{n \hookrightarrow q}) : \tau$ and $\tau \leq B(M)$ and $n_a \in M$, then there exist q_a and ρ_a such that $n_a \hookrightarrow q_a \in \overline{n \hookrightarrow q}$ and $\mathit{mtype}(q_a, C) = \rho_a$.

Proof. By induction on $\widehat{C}(\cdots) : \tau$.

case Tp-Subs. Immediate from induction hypothesis.

case Tp-New-Obj. By the form of the rule, $\tau = C(\overline{n : \rho})$, where $\mathit{mtype}(\overline{q}, C) = \overline{\rho}$. Let $M = \overline{n_m : \rho_m}$. By subtype and sub-row inversion (Lemmas A.28 and A.27), $\overline{n}_m \subseteq \overline{n}$. We have $n_a \in \overline{n}_m$. By the properties of the subset relation, $n_a \in \overline{n}$. Let k be the index of n_a in $\overline{n}$. Finally, taking ρ_k as ρ_a yields the required result. □

Lemma A.33 (Transitivity of the *super* judgement).
If Σ **ok** and $\Delta : \Sigma$ and:

1. $B \sqsubseteq_{\Delta} A$;
2. $A \sqsubseteq_{\Delta} C$ and $A \neq C$;
3. $C \neq \mathsf{Object}$; and
4. $\mathit{super}_{\Delta}(A \text{ as } C) = D$, for the unique result D,

then $\mathit{super}_{\Delta}(B \text{ as } C) = D$, for the unique result D.

Proof. By induction on $B \sqsubseteq_{\Delta} A$.

case Sub-Brand-Refl. Immediate.

case Sub-Brand-Trans. We have $B \sqsubseteq A'$ and $A' \sqsubseteq A$, for some A'. By Sub-Brand-Trans, $A' \sqsubseteq C$. Applying the induction hypothesis to the sub-derivation $A' \sqsubseteq A$ yields $\mathit{super}_{\Delta}(A' \text{ as } C) = D$, for some unique D. Taking this fact and applying the induction hypothesis to $B \sqsubseteq A'$ yields the required result.

case SUB-BRAND-DECL. We have B extends A. First, we will show that there does not exist D' such that $D' \sqsubseteq C$ and B extends D' (this will satisfy the first premise of SUPER-INH). Suppose such a D' did exist. Then B extends A, D' where $D' \sqsubseteq C$. But, by assumption, $A \sqsubseteq C$ and $C \neq$ Object, so this would violate the no-diamond property (Lemma. A.24). Therefore such a D' cannot exist. Taking A as E_k, we have $super_\Delta(B \text{ as } C) = D$ by SUPER-INH.

We must next show that there does *not* exist E_j such that B extends E_j and $super_\Delta(E_j \text{ as } C) = D''$, then we will have shown that the result of $super_\Delta(B \text{ as } C)$ is unique. Suppose such an E_j did exist. By Lemma A.23, we have $E_j \sqsubseteq D''$ and $D'' \sqsubseteq C$. From this it follows that $E_j \sqsubseteq C$ (by SUB-BRAND-TRANS). But, this would again create a diamond (with a brand other than Object at the top), as we have assumed B extends A, E_j and $E_j \sqsubseteq C$ and we have $A \sqsubseteq C$. □

Lemma A.34 (Conditions under which *super* is defined).

If Σ **ok** and $\Delta : \Sigma$ and $B \sqsubseteq_\Delta C$, where $B \neq C$ and $C \neq$ Object, then $super_\Delta(B \text{ as } C) = D$, for a unique result D.

Proof. By induction on $B \sqsubseteq_\Delta C$.

case SUB-BRAND-REFL. Vacuous, as we have assumed $B \neq C$.

case SUB-BRAND-TRANS. $B \sqsubseteq A \quad A \sqsubseteq C$

There are two subcases:

1. $B = A$ or $A = C$. Result follows from the induction hypothesis, as this gives a subderivation $B \sqsubseteq C$.
2. $B \neq A$ and $A \neq C$. By the induction hypothesis on $A \sqsubseteq C$, $super_\Delta(A \text{ as } C) = D$, for unique D. Taking this along with the assumptions $B \sqsubseteq A$ and $A \sqsubseteq C$, Lemma A.33 gives the required result.

case SUB-BRAND-DECL. B extends $C \in \Delta$

Take D as C. By SUB-BRAND-REFL, $C \sqsubseteq C$. The rule SUPER-BASE then applies. It now suffices to show that there does not exist D' such that B extends D' and $D' \sqsubseteq C$, as the result of $super_\Delta(B \text{ as } C)$ is then unique. The non-existence of such a D' follows from the no-diamond property (Lemma A.24), since we have $C \sqsubseteq C$ and we would have $D' \sqsubseteq C$. □

Lemma A.35 (Progress: expressions).

If Σ **ok** and $\cdot \vdash_\Sigma e : \tau$ then either e is a value, or, for any Δ such that $\Delta : \Sigma$, there exists e' such that $e \longmapsto_\Delta e'$.

Proof. By induction on $e : \tau$, with case analysis of final rule used.

case TP-UNIT, TP-FUN. Immediate.

case TP-APP. Straightforward.

case Tp-Subs. Result follows from induction hypothesis.

case Tp-New-Record. $e = (\overline{\ell = e})$
By the induction hypothesis, each e_i either steps to some e'_i or is a value. If any e_i steps, then the rule E-Record applies. Otherwise, the entire expression is a value.

case Tp-Proj. $e = e_1.\ell_k \qquad e_1 : \{\overline{\ell : \tau}\}$
By the induction hypothesis, either e_1 steps to some e'_1 or it is a value. If it steps to e'_1, then E-Proj1 applies. Otherwise, if it is a value, by canonical forms (Lemma A.30) e_1 has the form $(\overline{k = v})$ and E-Proj-Val applies.

case Tp-New-Obj. $e = \widehat{B}(e_1; \overline{n \hookrightarrow q})$
By the induction hypothesis, e_1 steps to some e'_1 or it is a value. If it takes a step, then E-Obj applies. If it is a value, then then e is also a value.

case Tp-With. $e = e_1 \;\mathsf{with}\; \overline{n = q}$
By the induction hypothesis, either e_1 is a value or it steps to some e'_1. If it steps, then the rule E-With applies. Otherwise, by canonical forms (Lemma A.30), e_1 has the form $\widehat{C}(v; \overline{n' \hookrightarrow q'})$. Then, the rule E-With-Val applies.

case Tp-Invoke-Struct, Tp-Invoke-Nom. $e = e_1.m \qquad e_1 : B(M)$
In either case, by the induction hypothesis, either e_1 steps to some e'_1 or it is a value. In the first case, E-Invoke applies. Otherwise, by canonical forms (Lemma A.30), e_1 has the form $\widehat{C}(v; \overline{n \hookrightarrow q})$. It suffices to show that $mbody_\Delta(m, C, \overline{n \hookrightarrow q})$ is uniquely defined; the rule E-Invoke-Val then applies.

case Tp-Invoke-Struct. $m = n$
By Lemma A.32, we have (a) $n \hookrightarrow q \in \overline{n \hookrightarrow q}$ and (b) $mtype_\Sigma(q, C) = \rho$, for some q and ρ. From (b), Lemma A.31 yields that $lookup_\Delta(q, C)$ is uniquely defined. From this and (a), the rule MBody-Simple applies. Since MBody-Qual is not applicable, *mbody* is also uniquely defined.

case Tp-Invoke-Nom. By Lemma A.31, $lookup_\Delta(q, C)$ is uniquely defined. The rule MBody-Qual then applies and *mbody* is therefore uniquely defined.

case Tp-Invoke-Super. $e = e_1.C.\mathsf{super}.q$
Either e_1 is a value or it evaluates to some e'_1. If it evaluates, the rule E-Super-Invk applies.

Otherwise, by canonical forms (Lemma A.30), e_1 has the form $\widehat{B}'(v; \overline{n \hookrightarrow q})$. By Lemma A.34, there exists a unique D such that $super_\Delta(B' \text{ as } C) = D$. By Lemma A.23 $B' \sqsubseteq_\Sigma D$. Precisifying the object type (Lemma A.17) gives us $\widehat{B}'(\cdots) : B'(M)$, and by Sub-Obj, $\widehat{B}'(\cdots) : D(M)$.

From the premise of the Tp-Invoke-Super, we have $mtype_\Sigma(q, C) = \rho$. Also, by Lemma A.23, $D \sqsubseteq_\Sigma C$. By Lemma A.21, $mtype_\Sigma(q, D) = \rho'$ (where $\rho' \le \rho$). Taking this fact together with $\widehat{B}'(\cdots) : D(M)$, by Lemma A.31, $lookup_\Delta(q, D)$ is uniquely defined. Then, E-Super-Invk-Val applies.

case Tp-Fold. $e = \mathsf{fold}_\tau\, e_1$

By the induction hypothesis, either e_1 takes a step or it is a value. If it takes a step, then the rule E-Fold applies. Otherwise, e itself is a value.

case Tp-Unfold $e = \mathsf{unfold}_{\mu X.\tau}\, e_1$
By the induction hypothesis, either e_1 takes a step or it is a value. If it takes a step, then the rule E-Unfold applies. Otherwise, it is a value v of type $\mu X.\tau$. By canonical forms (Lemma A.30), v has form $\mathsf{fold}_{\mu X.\tau}\, v_1$. so the rule E-Unfold-Fold applies. □

Theorem A.1 (Progress theorem for programs [Theorem 3.2]).
If Σ **ok** and $\Sigma \vdash p$ **ok**, then one of the following cases holds:

1. p is a value; or
2. for any Δ such that $\Delta : \Sigma$, there exist p' and Δ' such that $p \mid \Delta \longmapsto p' \mid \Delta'$.

Proof. By case analysis of the form of p.

case $p = \Sigma \triangleright \mathit{decl}$ in p.
Either E-Brand-Decl or E-Ext-Decl applies.

case $p = \Sigma \triangleright e$.
The result follows from the progress lemma for expressions (Lemma A.35) and E-Expr. □

A.5 Preservation Lemmas and Theorem

Lemma A.36 (Weakening for Γ).
If Σ **ok** and $\Gamma \vdash_\Sigma e : \tau$, then $\Gamma, x : \sigma \vdash_\Sigma e : \tau$

Proof. Straightforward. □

Lemma A.37 (Substitution).
If Σ **ok** and $\Gamma, x : \sigma \vdash_\Sigma e_1 : \tau$ and $\Gamma \vdash_\Sigma e_2 : \sigma$ then $\Gamma \vdash_\Sigma \{e_2/x\}\, e_1 : \tau$.

Proof. Straightforward induction on typing derivations. □

Lemma A.38. If Σ **ok** and $C \sqsubseteq_\Sigma D$, then $\text{fieldWithReq}_\Sigma(C) \leq \text{fieldWithReq}_\Sigma(D)$.

Proof. Straighforward induction on $C \sqsubseteq D$. For the base case, by Lemma A.14, we have $\vdash_\Sigma$ $\mathsf{brand}\ C\ \mathsf{extends}\ D$ **ok**. Inversion on this judgement yields premises (1) and (3) of Tp-Brand-Decl, which give the required result. □

Lemma A.39 (*mtype* is a function [Lemma 3.5]).
If Σ **ok** and $\mathit{mtype}_\Sigma(q, B) = \rho_1$ and $\mathit{mtype}_\Sigma(q, B) = \rho_2$, then $\rho_1 = \rho_2$.

Proof. By simultaneous induction on the two *mtype* derivations, making use of the fact that MType-Inh excludes MType-Base and MType-Ext.

case MType-Base, MType-Base. Immediate from the property of non-duplicate internal method names (Lemma A.25).

case MType-Base, MType-Ext. Vacuous: external methods may not override internal methods. That is, Lemma A.26 gives $\nvdash_\Sigma B.q$ **internal**, but MType-Base assumes $q \in B$, i.e., $\vdash_\Sigma B.q$ **internal**.

case MType-Ext, MType-Ext. Immediate from the properties of unique method external method families and non-duplicate external method definitions (Lemmas A.1 and A.26).

case MType-Inh, MType-Inh. We have $B\,\mathsf{extends}\,C_1 \in \Sigma$ and $B\,\mathsf{extends}\,C_2 \in \Sigma$. Let $\mathit{mtype}_\Sigma(q, C_1) = \rho_1$ and $\mathit{mtype}_\Sigma(q, C_2) = \rho_2$.

But, we will now show that $C_1 = C_2$. By Lemma A.36, $\Sigma \vdash B$ **ok**. By inversion on this derivation, we have premise (1) of Tp-Brand-Decl, i.e., $\vdash_\Sigma B$ **inherit-ok**. Finally, by inversion on this last derivation, we have premise (7) of Tp-Inherit, which requires that *mtype*(q) derivations do not exist for two distinct superclasses.

Now, by the induction hypothesis, $\rho_1 = \rho_2$. The result then follows from MType-Inh. □

Lemma A.40 (Result of *lookup* is well-typed [Lemma 3.6]).
If Σ **ok** and $\Delta : \Sigma$ and $\mathit{mtype}_\Sigma(q, C) = N \Rightarrow \tau$ and $\mathit{lookup}_\Delta(q, C) = e_0$, then
$\mathsf{this} : \sigma_c, \mathsf{fields} : \sigma_f \vdash_\Sigma e_0 : \tau$.
for some σ_c and σ_f such that $C(N) \le \sigma_c$ and $\mathrm{fieldWithReq}_\Sigma(C) \le \sigma_f$.

Proof. By induction on $\mathit{lookup}_\Delta(q, C)$.

case Lookup-Base. By the definition of $\Delta : \Sigma$, q must be defined internally in C with some type $\rho = N \Rightarrow \tau$. By MType-Base, $\mathit{mtype}_\Sigma(q, C) = N \Rightarrow \tau$ (for the unique result $N \Rightarrow \tau$, since by Lemma A.25, internal method names q are distinct from one another). Finally, from the definition of $\Delta : \Sigma$ (Def. 3.3), we have:
$\mathsf{this} : C(N), \mathsf{fields} : \mathrm{fieldWithReq}_\Sigma(C) \vdash_\Sigma e_0 : \tau$,
which is the required result.

case Lookup-Ext. By the definition of $\Delta : \Sigma$, q must be defined externally for C with some type $\rho = N \Rightarrow \tau$. By the properties of well-formed external methods (Lemma A.26), $\nvdash C.q$ **internal**. By the definition of **internal**, q is not defined internally in C, so the rule MType-Base cannot apply. By inspection, the rule MType-Ext applies and $\mathit{mtype}_\Sigma(q, C) = N \Rightarrow \tau$. Finally, Lemmas A.1 and A.26, there is only one method family q and there cannot be a duplicate entry for C; therefore, the result $N \Rightarrow \tau$ is unique.

From the definition of $\Delta : \Sigma$ (Def. 3.3), we have:
$\mathsf{this} : C(N) \vdash_\Sigma e_0 : \tau$.
Applying weakening for Γ (Lemma A.36) gives the required result.

case Lookup-Inh. By the premises of the rule, q is not defined internally or externally on C. Also, $C\,\mathsf{extends}\,D \in \Delta$ and $\mathit{lookup}_\Delta(q, D) = e_0$. By Lemma A.22, the derivation $\mathit{lookup}_\Delta(q, D)$ implies that there exists a derivation $\mathcal{D} :: \mathit{mtype}_\Sigma(q, D) = \rho$, for some ρ and some $\mathcal{D}$ that does not contain MType-Req. By MType-Inh, $\mathcal{D}_2 :: \mathit{mtype}_\Sigma(q, C) = \rho$, where $\mathcal{D}_2$ does not contain MType-Req. By Lemma A.39, ρ is the same as the result of the assumed *mtype* derivation; that is, $\rho = N \Rightarrow \tau$.

By the induction hypothesis, we have:

$\mathsf{this} : \sigma_d, \mathsf{fields} : \sigma_f \vdash_\Sigma e_0 : \tau$,

for σ_d such that $D(N) \le \sigma_d$ and σ_f such that $\mathrm{fieldWithReq}_\Sigma(D) \le \sigma_f$.

It remains to show that (a) $C(N) \le \sigma_d$ and (b) $\mathrm{fieldWithReq}_\Sigma(C) \le \sigma_f$. Result (a) follows from Sub-Brand-Decl, Sub-Obj, and Sub-Trans. Result (b) follows from Lemma A.38 (i.e., $\mathrm{fieldWithReq}_\Sigma(C) \le \mathrm{fieldWithReq}_\Sigma(D)$) and Sub-Trans. □

Lemma A.41. If Σ **ok** and $C \sqsubseteq_\Sigma B$ and B requires $D \in \Sigma$, then either $C \sqsubseteq_\Sigma D$ or C requires D', for D' such that $D' \sqsubseteq_\Sigma D$.

Proof. Straightforward induction on $C \sqsubseteq_\Sigma B$, using Lemma A.14 and inversion of **inherit-ok** in the base case. □

Lemma A.42 (Preservation: expressions).
If Σ **ok** and $\Gamma \vdash_\Sigma e : \tau$ and $\Delta : \Sigma$ and $e \longmapsto_\Delta e'$, then $\Gamma \vdash_\Sigma e' : \tau$.

Proof. By induction on $e : \tau$.

case Tp-Var, Tp-Unit, Tp-Fun. Vacuous; e does not evaluate.

case Tp-App. Straightforward, using Lemma A.37.

case Tp-Subs. $e : \sigma \quad \sigma \le \tau \quad e \longmapsto_\Delta e'$
By the induction hypothesis, $e' : \sigma$ and the result follows from Tp-Subs.

case Tp-New-Record. The only evaluation rule that applies is E-Record. We have $e_k \longmapsto_\Delta e'_k$. By the induction hypothesis, $e'_k : \tau_k$. The result then follows from Tp-New-Record.

case Tp-Proj. $e : \{k_i : \tau_i{}^{i \in 1..n}\}$
There are two possible evaluation rules that apply:

subcase E-Proj1. Result follows from the induction hypothesis and Tp-Proj.

subcase E-Proj2. $(\ell_j = v_j{}^{j \in 1..m}).\ell_k \longmapsto_\Delta v_k$
By typing inversion (Lemma A.29), we have $\{\ell_j : \tau_j{}^{j \in 1..m}\} \le \{k_i : \tau_i{}^{i \in 1..n}\}$ and $v_k : \tau_k$, which is the required result.

case Tp-New-Obj. $e = \widehat{B}(e_1; \overline{n \hookrightarrow q}) \qquad e_1 : \tau' \qquad e_1 \longmapsto_\Delta e'_1$
The only evaluation rule that is applicable is E-Obj. By the induction hypothesis, $e'_1 : \tau'$. The result then follows from Tp-New-Obj.

case Tp-With.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : B(M) \qquad B \text{ requires } \overline{D} \in \Sigma \\ \overline{n}_b \notin M \qquad \overline{n}_b \text{ distinct} \qquad \exists C' \in \{B, \overline{D}\}.\, \mathit{mtype}_\Sigma(q_{b_i}, C') : \rho_{b_i} \quad (\forall i \in 1..n)\end{array}}{\Gamma \vdash e_1 \text{ with } n_{b_i} \hookrightarrow q_{b_i}{}^{i \in 1..n} : B(M, n_{b_i} : \rho_{b_i}{}^{i \in 1..n})}$$

There are two possible evaluation rules that apply:

subcase E-With. Result follows from the induction hypothesis and Tp-With.

subcase E-With-Val.

$$\widehat{C}(v'; \overline{n_a \hookrightarrow q_a}) \text{ with } \overline{n_b \hookrightarrow q_b} \longmapsto_\Delta \widehat{C}(v';\ \mathit{merge}(\overline{n_b \hookrightarrow q_b}, \overline{n_a \hookrightarrow q_a}))$$

$$v_1 = e_1 = \widehat{C}(v'; \overline{n_a \hookrightarrow q_a}) \qquad v_1 : B(\overline{n_m : \rho_m}) \qquad M = \overline{n_m : \rho_m}$$

Precisifying the type of v_1 (Lemma A.17) yields:
(1) $v_1 : C(\overline{n_a : \rho_a})$
(2) $C(\overline{n_a : \rho_a}) \leq B(\overline{n_m : \rho_m})$, where $\mathit{mtype}_\Sigma(\overline{q}_a, C) = \overline{\rho}_a$, and
(3) C requires $\bullet \in \Sigma$.

By inversion on the subtyping derivation (2) (Lemma A.28),
(i) $C \sqsubseteq B$ and
(ii) $\overline{n_a : \rho_a} \trianglelefteq \overline{n_m : \rho_m}$

By sub-row inversion on (ii) (Lemma A.27):
$\rho_{a_j} \leq \rho_{m_i}$ for each commom label $n_{a_j} = n_{m_i}$
$\overline{n}_m \subseteq \overline{n}_a$

Let $\overline{n}_{a'}$ be the set of labels that are in $\overline{n}_a$ but not in $\overline{n}_b$ (that is, $\overline{n}_{a'} = \overline{n}_a - \overline{n}_b$, where "−" is set difference). By the definition of *merge*, $\mathit{merge}(\overline{n_b \hookrightarrow q_b}, \overline{n_a \hookrightarrow q_a}) = \overline{n_b \hookrightarrow q_b}, \overline{n_{a'} \hookrightarrow q_{a'}}$.

Let $v = \widehat{C}(v';\ \mathit{merge}(\overline{n_b \hookrightarrow q_b}, \overline{n_a \hookrightarrow q_a}))$. From above, $v = \widehat{C}(v';\ \overline{n_b \hookrightarrow q_b}, \overline{n_{a'} \hookrightarrow q_{a'}})$. We are to show that $v : B(M, \overline{n_b : \rho_b})$, i.e., $v : B(\overline{n_m : \rho_m}, \overline{n_b : \rho_b})$.

By premise (5) of Tp-With, for all $i \in 1..n$, either (a) $\mathit{mtype}_\Sigma(q_{b_i}, B) : \rho_{b_i}$ or (b) $\exists k.\, \mathit{mtype}_\Sigma(q_{b_i}, D_k) : \rho_{b_i}$. By Lemma A.41, since $C \sqsubseteq B$ and C requires $\bullet$, $C \sqsubseteq \overline{D}$. For each i, in either case, by Lemma A.21, $\mathit{mtype}_\Sigma(q_{b_i}, C) : \rho'_{b_i}$, where $\rho'_{b_i} \leq \rho_{b_i}$.

Applying Tp-Obj, $v : C(\overline{n_b : \rho'_b}, \overline{n_{a'} : \rho_{a'}})$ By premise (3) of Tp-With, $\overline{n}_b, \overline{n}_m$ are disjoint. From this, we can conclude $\overline{n}_m \subseteq \overline{n}'_a$. Since we have $\rho_{a'_j} \leq \rho_{m_i}$ for $n_{a'_j} = n_{m_i}$, by Sub-Row-Width and Sub-Row-Depth, $\overline{n_{a'} : \rho_{a'}} \trianglelefteq \overline{n_m : \rho_m}$. Also, by Sub-Row-Depth, $\overline{n_b : \rho'_b} \trianglelefteq \overline{n_b : \rho_b}$. Finally, by Sub-Row-Perm, $\overline{n_b : \rho'_b}, \overline{n_{a'} : \rho_{a'}} \trianglelefteq \overline{n_m : \rho_m}, \overline{n_b : \rho_b}$, which is the structural part of the required result. Finally, since $C \sqsubseteq B$, the result follows from Sub-Obj and Tp-Subs.

case Tp-Invoke-Struct.

$$\frac{\Gamma \vdash e_1 : B(M) \qquad n : N \Rightarrow \tau \in M \qquad M \trianglelefteq N}{\Gamma \vdash e_1.n : \tau}$$

There are two possible evaluation rules that apply:

subcase E-Invoke. Result follows from the induction hypothesis and Tp-Invoke.

subcase E-Invoke-Val.

$$\frac{\exists \text{ unique } e_0.\ \mathit{mbody}_\Delta(n, C, \overline{n \hookrightarrow q}) = e_0}{\widehat{C}(v'; \overline{n \hookrightarrow q}).n \longmapsto_\Delta \{\widehat{C}(v'; \overline{n \hookrightarrow q})/\mathsf{this}, v'/\mathsf{fields}\}\ e_0}$$

$$e = v_1.n \qquad e_1 = v_1 = \widehat{C}(v'; \overline{n \hookrightarrow q}) \qquad v_1 : B(M)$$

By precisifying v_1 (Lemma A.17):
$v_1 : C(\overline{n : \rho})$,
and $C(\overline{n : \rho}) \leq B(M)$ and C $\mathsf{requires}$ $\bullet$, where $\mathit{mtype}(\overline{q}) = \overline{\rho}$. Applying Lemma A.29 (inversion on the typing derivation) to $v_1 : C(\overline{n : \rho})$, we have:
$v' : \text{fieldType}_\Sigma(C)$.

It suffices to show that:
$\mathsf{this} : \tau_c, \mathsf{fields} : \tau_f \vdash e_0 : \tau$,
for some τ_c and τ_f where $C(\overline{n : \rho}) \leq \tau_c$ and $\text{fieldType}_\Sigma(C) \leq \tau_f$, the result then follows from the substitution lemma (Lemma A.37).

By MBody-Simple, $\mathit{mbody}_\Delta(n, C, \overline{n \hookrightarrow q}) = \mathit{lookup}_\Delta(q, C)$, where $n \hookrightarrow q \in \overline{n \hookrightarrow q}$. From above (result of Lemma A.17), $\mathit{mtype}_\Sigma(q, C) = \rho$. Let $\rho = N' \Rightarrow \tau'$.

Applying Lemma A.40, we have:
$\mathsf{this} : \sigma_c, \mathsf{fields} : \sigma_f \vdash_\Sigma e_0 : \tau$,
for some σ_c and σ_f such that $C(N') \leq \sigma_c$ and $\text{fieldWithReq}_\Sigma(C) \leq \sigma_f$.

Recall from above that C $\mathsf{requires}$ $\bullet$. From this, it follows that $\text{fieldWithReq}_\Sigma(C) = \text{fieldType}_\Sigma(C)$ and we can take σ_f as τ_f from above.

It now suffices to show that (a) $C(\overline{n : \rho}) \leq C(N')$ and (b) $\tau' \leq \tau$; then we may take σ_c as τ_c and apply Tp-Subs. By subtype inversion (Lemma A.28) on $C(\overline{n : \rho}) \leq B(M)$ we have $\overline{n : \rho} \trianglelefteq M$. By Tp-Sub-Row, $\overline{n : \rho} \trianglelefteq N$. Also, since we have $n : N \Rightarrow \tau \in M$, by subrow inversion (Lemma A.27) we also have $N' \Rightarrow \tau' \leq N \Rightarrow \tau$. By subtype inversion (Lemma A.28) on this last derivation, $N \trianglelefteq N'$ and $\tau' \leq \tau$, the latter satisfying (b).

Finally, by Sub-Row-Trans, $\overline{n : \rho} \leq N'$. Tp-Obj yields $C(\overline{n : \rho}) \leq C(N')$ which is the required result.

case Tp-Invoke-Nom. $e = e_1.q$

$$\frac{\Gamma \vdash e_1 : B(M) \qquad \mathit{mtype}_\Sigma(q, B) = N \Rightarrow \tau \qquad M \trianglelefteq N}{\Gamma \vdash e_1.q : \tau}$$

There are two possible evaluation rules that apply:

subcase E-Invoke. Result follows from the induction hypothesis and Tp-Invoke.

subcase E-Invoke-Val.

$$\frac{\exists \text{ unique } e_0.\, \mathit{mbody}_\Delta(q, C, \overline{n \hookrightarrow q}) = e_0}{\widehat{C}(v'; \overline{n \hookrightarrow q}).q \longmapsto_\Delta \{\widehat{C}(v'; \overline{n \hookrightarrow q})/\mathsf{this}, v'/\mathsf{fields}\}\, e_0}$$

$$e = v_1.q \qquad e_1 = v_1 = \widehat{C}(v'; \overline{n \hookrightarrow q}) \qquad v_1 : B(M)$$

This proof is very similar to that of case Tp-Invoke-Struct, E-Invoke-Val above. The main difference is the reasoning for obtaining a derivation $\mathit{mtype}_\Sigma(q, C) = N' \Rightarrow \tau'$.

By precisifying v_1 (Lemma A.17):
$v_1 : C(\overline{n : \rho})$,
and $C(\overline{n : \rho}) \leq B(M)$ and C $\mathsf{requires}$ $\bullet$. Applying Lemma A.29 (inversion on the typing derivation) to $v_1 : C(\overline{n : \rho})$, we have:
$v' : \text{fieldType}_\Sigma(C)$.

It suffices to show that:
$\mathsf{this} : \tau_c, \mathsf{fields} : \tau_f \vdash e_0 : \tau$,
for some τ_c and τ_f where $C(\overline{n : \rho}) \leq \tau_c$ and $\text{fieldType}_\Sigma(C) \leq \tau_f$, the result then follows from the substitution lemma (Lemma A.37).

By a straightforward inversion on the derivation $\mathit{mbody}_\Delta(q, C, \cdots)$, we have $\mathit{lookup}_\Delta(q, C) = e_0$. From this, Lemma A.22 there is a derivation $\mathit{mtype}_\Sigma(q, C) = N' \Rightarrow \tau'$, for some $N' \Rightarrow \tau'$.

Applying Lemma A.40, we have:
$\mathsf{this} : \sigma_c, \mathsf{fields} : \sigma_f \vdash_\Sigma e_0 : \tau$,
for some σ_c and σ_f such that $C(N') \leq \sigma_c$ and $\text{fieldWithReq}_\Sigma(C) \leq \sigma_f$.

Recall from above that C $\mathsf{requires}$ $\bullet$. From this, it follows that $\text{fieldWithReq}_\Sigma(C) = \text{fieldType}_\Sigma(C)$ and we can take σ_f as τ_f from above.

It now suffices to show that (a) $C(\overline{n : \rho}) \leq C(N')$ and (b) $\tau' \leq \tau$; then we may take σ_c as τ_c. By inversion on the subtype derivation $C(\overline{n : \rho}) \leq B(M)$ above (Lemma A.28), we have $\overline{n : \rho} \trianglelefteq M$ and $C \sqsubseteq_\Sigma B$ (since C $\mathsf{requires}$ $\bullet$). By Sub-Row-Trans, $\overline{n : \rho} \trianglelefteq N$.

Recall that $\mathit{mtype}_\Sigma(q, B) = N \Rightarrow \tau$ and $\mathit{mtype}_\Sigma(q, C) = N' \Rightarrow \tau'$. By Lemma A.21, $N \trianglelefteq N'$ and $\tau' \leq \tau$ (the latter satisfying (b) above). Taking this together with $\overline{n : \rho} \trianglelefteq N$ and applying Sub-Row-Trans, Sub-Obj and Tp-Subs, we have $C(\overline{n : \rho}) : C(N')$, which is the required result.

case Tp-Invoke-Super. There are two possible evaluation rules that apply:

subcase E-Super-Invk. Result follows from the induction hypothesis and Tp-Invoke-Super.

subcase E-Super-Invk-Val.

$$\frac{\exists \text{ unique } D.\, \mathit{super}_{\Delta}(B \text{ as } C') = D \qquad \exists \text{ unique } e_0.\, \mathit{lookup}_{\Delta}(q, D) = e_0}{\widehat{C}(v'; \overline{n \hookrightarrow q}).C.\mathsf{super}.q \longmapsto_{\Delta} \{\widehat{C}(v; \overline{n \hookrightarrow q})/\mathsf{this}, v'/\mathsf{fields}\}\ e_0}$$

$e \longmapsto_{\Delta} e'$
$e = v_1.C'.\mathsf{super}.q \qquad v_1 = \widehat{C}(v'; \overline{n \hookrightarrow q}) \qquad v_1 : B(M)$
$B \text{ requires } C' \in \Sigma \qquad \mathit{mtype}_{\Sigma}(q, C') = N \Rightarrow \tau \qquad M \trianglelefteq N$

By Lemma A.23, $C \sqsubseteq D$ and $D \sqsubseteq C'$.

By Lemma A.21, since $D \sqsubseteq C'$, we have $\mathit{mtype}_{\Sigma}(q, D) = N' \Rightarrow \tau'$, where $N' \Rightarrow \tau' \leq N \Rightarrow \tau$. Again by Lemma A.21, since $C \sqsubseteq D$, $\mathit{mtype}_{\Sigma}(q, C) = N'' \Rightarrow \tau''$ where $N'' \Rightarrow \tau'' \leq N' \Rightarrow \tau'$. Applying subtyping inversion to these judgements (Lemma A.28), subrow inversion and transitivity yield $N \trianglelefteq N''$ and $\tau'' \leq \tau$.

Recall that we have $\mathit{lookup}_{\Delta}(q, D) = e_0$ and $\mathit{mtype}_{\Sigma}(q, D) = N' \Rightarrow \tau'$. Applying Lemma A.40, we have $\mathsf{this} : \sigma_c, \mathsf{fields} : \sigma_f \vdash_{\Sigma} e_0 : \tau$, for some σ_c and σ_f such that $D(N') \leq \sigma_c$ and $\text{fieldWithReq}_{\Sigma}(D) \leq \sigma_f$.

From $C \sqsubseteq D$ and $N'' \trianglelefteq N$, Sub-Obj yields $C(N'') \leq D(N')$; by transitivity, $C(N'') \leq \sigma_c$. By Lemma A.38, $\text{fieldWithReq}_{\Sigma}(C) \leq \text{fieldWithReq}_{\Sigma}(D)$, which by transitivity yields $\text{fieldWithReq}_{\Sigma}(C) \leq \sigma_f$.

By the subsitution lemma (Lemma A.37), $e : \tau$. Finally, since $\tau'' \leq \tau$, Tp-Subs yields the required result.

case Tp-Fold. The only evaluation rule that applies is E-Fold. The result follows from the induction hypothesis and Tp-Fold.

case Tp-Unfold. There are two possible evaluation rules that apply:

subcase E-Unfold. The result follows from the induction hypothesis and Tp-Unfold.

subcase E-Unfold-Fold.
Straightforward, using typing inversion lemma (Lemma A.29). □

Theorem A.2 (Preservation theorem for programs [Theorem 3.3]).
If Σ **ok** and $\Sigma \vdash p$ **ok** and $\Delta : \Sigma$ and $p \mid \Delta \longmapsto p' \mid \Delta'$, then there exists a Σ' such that (a) Σ' **ok** and (b) $\Delta' : \Sigma'$ and (c) $\Sigma' \vdash p'$ **ok**.

Proof. By case analysis on the derivation $\Sigma \vdash p$ **ok**.

case Tp-Decl-Ok.

$$\frac{\begin{array}{c} \textcircled{1}\ \mathit{decl} : \mathit{decl\text{-}type} \qquad \textcircled{2}\ \mathit{decl\text{-}type}_{\mathit{name}} \notin \Sigma \qquad \textcircled{3}\ \Sigma_0 \vdash \mathit{decl\text{-}type}\ \mathbf{ok} \\ \textcircled{4}\ \Sigma_0, \mathit{decl\text{-}type} \vdash \mathit{decl}\ \mathbf{body\text{-}ok} \qquad \textcircled{5}\ \Sigma \supseteq \Sigma_0 \qquad \textcircled{6}\ \Sigma, \mathit{decl\text{-}type} \vdash p'\ \mathbf{ok} \end{array}}{\Sigma \vdash \Sigma_0 \triangleright \mathit{decl}\ \mathsf{in}\ p'\ \mathbf{ok}}$$

By E-Brand-Decl and E-Ext-Decl, $p \mid \Delta \longmapsto p' \mid \Delta'$. Let $\Sigma' = \Sigma, \textit{decl-type}$.

By premises (3) and (5), $\Sigma_0 \vdash \textit{decl-type}\ \mathbf{ok}$ and $\Sigma \supseteq \Sigma_0$. By weakening on Σ_0 (Lemma A.36), $\Sigma \vdash \textit{decl-type}\ \mathbf{ok}$. This, together with premise (2), yields $\Sigma'\ \mathbf{ok}$, which is part (a) of the required result.

Part (c) of the required result, $\Sigma' \vdash p'\ \mathbf{ok}$ is immediate from premise (6).

To prove part (b), $\Delta' : \Sigma'$, we case analyze the form of *decl-type.*

subcase Brand declaration.

$\textit{decl} = \mathsf{brand}\ B(\sigma; q_i\ B(M_i) : \tau_i = e_i\ ^{i \in 1..n})\ \mathsf{extends}\ \overline{C}\ \mathsf{requires}\ \overline{D}$
$\textit{decl-type} = \mathsf{brand}\ B(\sigma; \overline{q : M \Rightarrow \tau})\ \mathsf{extends}\ \overline{C}\ \mathsf{requires}\ \overline{D}$
$\Delta' = \Delta, B(\overline{q = e})\ \mathsf{extends}\ \overline{C}$

Let $\Sigma'_0 = \Sigma_0, \textit{decl-type}$. By assumption, $\Sigma'_0 \vdash \textit{decl}\ \mathbf{body\text{-}ok}$. Inversion on this derivation yields (by Brand-Decl-Body):

(i) $\mathrm{fieldType}_{\Sigma'_0}(\overline{D}) = \overline{\sigma}'$

(ii) $\mathsf{this} : B(M_i), \mathsf{fields} : \sigma \wedge \overline{\sigma}' \vdash_{\Sigma'_0} e_i : \tau_i$, for all $i \in 1..n$

It suffices to show that derivations (i) and (ii) hold under the larger context Σ', as then Delta-Wf-Brand yields the required result.

By the properties of set inclusion, $\Sigma \supseteq \Sigma_0$ implies that $\Sigma' \supseteq \Sigma'_0$.

Since $\Sigma'\ \mathbf{ok}$, there are no duplicate brands in Σ' (Lemma A.1). Therefore, item (i) has the same result under context Σ' (i.e., $\overline{\sigma}'$).

For item (ii), we use the fact that $\Sigma'\ \mathbf{ok}$ (proved above for part (a) of the theorem) and apply signature weakening for expression typing (Lemma A.15). This yields

$\mathsf{this} : B(M_i), \mathsf{fields} : \mathrm{fieldWithReq}_{\Sigma}(B) \vdash_{\Sigma} e_i : \tau_i$, for all $i \in 1..n$,

which is the required result.

subcase External method declaration.

Similar to above; inversion on $\Sigma'_0 \textit{decl-type}\ \mathbf{body\text{-}ok}$ yields the sole premise of Ext-Method-Body (which is similar to (ii) above, except the special variable fields is not bound in Γ. The result follows from the properties of set inclusion and Delta-Wf-Method.

case Tp-Expr-Ok.

The rule E-Expr is the only rule that applies, i.e., $e \longmapsto_{\Delta} e'$. By the preservation lemma for expressions (Lemma A.42), $\vdash_{\Sigma_0} e' : \tau$. Since $\Delta' = \Delta$, take $\Sigma' = \Sigma$; the result then follows from Tp-Expr-Ok. □

Theorem A.3 (Type safety).

If $\Sigma\ \mathbf{ok}$ and $\Sigma \vdash p\ \mathbf{ok}$ and $\Delta : \Sigma$, then either (1) p is a value or (2) $p \mid \Delta \longmapsto p' \mid \Delta'$, for some p' and Δ' where $\Delta' : \Sigma'$ and $\Sigma'\ \mathbf{ok}$ and $\Sigma' \vdash p'\ \mathbf{ok}$.

Proof. Follows from progress and preservation theorems (Theorems A.1 and A.2). □

Corollary A.1 (Method lookup always succeeds in well-typed expressions).
If Σ **ok** and $\vdash_\Sigma\ v.m : \tau$ and $\Delta : \Sigma$ then $v.m \longmapsto_\Delta e'$, for some unique e'.

Proof. Follows from progress lemma on expressions (Lemma A.35) and the sole premise of the method evaluation rule E-Invoke-Val, which requires that the result of method body lookup (*mbody*) be uniquely defined. □

Theorem A.4 (Typechecking is modular [Theorem 3.1]).
Typechecking top-level elements declarations *decl* is modular. That is, typechecking such elements only involves examining the signatures on which *decl* statically depends.

Proof. Follows from the fact that top-level elements are typechecked under their declared context Σ_0. The only rules that examine the entire linearized program context Σ are Tp-Decl-Ok and Tp-Expr-Ok, in the premise $\Sigma \supseteq \Sigma_0$. This step is analogous to a linking phase in which imported declarations are resolved. Since checking set inclusion does not involve typechecking of any kind, this check adheres to the definition of modular typechecking. □

Appendix B

Empirical Study Details

B.1 Subject Programs

Program	Version
Ant	1.7.0
antlr	2.7.6
Apache collections	3.2
Areca	5.5.3
Cayenne	2.0.4
Columba	1.0RC1
Crystal	3.3.0
DrJava	20080904-r4668
Emma	2.0.5312
freecol	0.7.3
hsqldb	1.8.0.4
HttpClient	3.1
jEdit	4.2
JFreeChart	1.0.0-rc1
JHotDraw	7.0.9
jruby	1.0.1
jung	1.7.6
LimeWire	4.13.0
log4j	1.2.15
Lucene	1.4
OpenFire	3.4.2
plt collections	20080904-r4668
pmd	3.3
poi	2.5.1
quartz	1.5.2
Smack	3.0.4
Struts	2.0.11
Tomcat	6.0.14
xalan	2.7.0

Table B.1: Version numbers of empirical study subject programs

B.2 Subjective Criteria

In Section 4.3.2, I enumerated the number of cases where it could be "useful" to generalize the parameter types of a particular method. To determine this, I asked two questions. First, does the inferred parameter type S generalize the abstract operation performed by the method (as determined by the method name)? For example, generalizing the List parameters in ListUtils.intersection *does* appear to generalize the abstract operation of taking the intersection of two sequences. Second, does it seem likely that there would be multiple subtypes of S? For example, in Crystal I found that there were two methods of the IBinding interface that were often used, and I was informed by the developers that it was conceivable that they would replace the use of Eclipse binding objects with an application-specific representation.

In Section 4.5.2, I tabulated the number of methods in a common method group that had "the same meaning." To determine this, I used javadoc when available; when it was not, I examined the body of the method to determine the operation being performed.

Bibliography

Agrawal, R., L. DeMichiel, and B. Lindsay (1991). Static type checking of multi-methods. In *OOPSLA*, pp. 113–128. 119

Allen, E., D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele, Jr., and S. Tobin-Hochstadt (2008). The Fortress Language Specification, Version 1.0. Available at http://research.sun.com/projects/plrg/Publications/fortress.1.0.pdf. 6, 9, 69

Allen, E., J. J. Hallett, V. Luchangco, S. Ryu, and G. L. Steele Jr. (2007). Modular multiple dispatch with multiple inheritance. In *SAC '07*, pp. 1117–1121. ACM. 7, 63, 65, 69, 90

Ancona, D., G. Lagorio, and E. Zucca (2003). Jam - designing a Java extension with mixins. *ACM Trans. Program. Lang. Syst. 25*(5), 641–712. 6, 54, 76, 77, 90

Ancona, D. and E. Zucca (1996). An algebraic approach to mixins and modularity. In *Algebraic and Logic Programming*, pp. 179–193. 23, 54, 90, 113

Baldi, P., C. Lopes, E. Linstead, and S. Bajracharya (2008). A theory of aspects as latent topics. In *OOPSLA*. 119

Bank, J., A. Myers, and B. Liskov (1997). Parameterized types for Java. In *POPL '97*, pp. 132–145. 114

Barnett, M., B. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino (2005). Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, pp. 364–387. 127

Barnett, M., R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte (2004). Verification of object-oriented programs with invariants. *Journal of Object Technology 3*(6), 27–56. 127

Baumgartner, G., M. Jansche, and K. Läufer (2002, March). Half & Half: Multiple dispatch and retroactive abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Dept. of Computer and Information Science, The Ohio State University. 119

Baumgartner, G. and V. Russo (1997). Implementing signatures for C++. *ACM Trans. Program. Lang. Syst. 19*(1), 153–187. 114

Beaven, M. and R. Stansifer (1993). Explaining type errors in polymorphic languages. *ACM Lett. Program. Lang. Syst. 2*(1-4), 17–30. 126

Bergel, A. (2008, October). Personal communication. 65

Bergel, A., S. Ducasse, O. Nierstrasz, and R. Wuyts (2008). Stateful traits and their formalization.

Computer Languages, Systems & Structures 34(2-3), 83–108. 6, 64, 65, 77, 91

Bettini, L., V. Bono, and S. Likavec (2004). A core calculus of higher-order mixins and classes. In *SAC*, pp. 1508–1509. 54, 90

Bierhoff, K. and J. Aldrich (2005). Lightweight object specification with typestates. In *ESEC/FSE-13*, pp. 217–226. 127

Bierhoff, K. and J. Aldrich (2007, October). Modular typestate checking of aliased objects. In *OOPSLA '07*, pp. 301–320. 127

Black, A., N. Hutchinson, E. Jul, and H. Levy (1986). Object structure in the emerald system. In *OOPLSA '86*, pp. 78–86. 73

Bloch, J. (2001). *Effective Java: Programming Language Guide.* Addison-Wesley. 79, 95, 96

Bobrow, D., L. DeMichiel, R. Gabriel, S. Keene, G. Kicsales, and D. Moon (1988). Common LISP object system specification. *ACM SigPLAN Notices 23*, 1–143. 119

Bonniot, D. (2007). The Nice programming language. Available at http://nice.sourceforge.net. Accessed 8/09. 116, 117

Bono, V., F. Damiani, and E. Giachino (2007). Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse. In *Electronic proceedings of FTfJP'07 (http://www.cs.ru.nl/ftfjp/).* 114

Boyland, J. and G. Castagna (1997). Parasitic methods: An implementation of multi-methods for Java. In *OOPSLA*, pp. 66–76. 119

Bracha, G. (2006, March). Personal communication. 126

Bracha, G. and W. Cook (1990). Mixin-based inheritance. In *ECOOP '90*. 6, 23, 54, 61, 90, 113

Bracha, G. and D. Griswold (1993). Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA '93*, pp. 215–230. 7, 114, 126

Bruce, K., A. Schuett, R. van Gent, and A. Fiech (2003). Polytoil: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst. 25*(2), 225–290. 7, 55, 93, 113

Büchi, M. and W. Weck (1998). Compound types for Java. In *OOPSLA '98*, pp. 362–373. 4, 52, 113

Cardelli, L. (1988). Structural subtyping and the notion of power type. In *POPL '88*, pp. 70–79. 7, 93, 113

Carré, B. and J. Geib (1990). The point of view notion for multiple inheritance. In *OOPSLA/ECOOP '90*, pp. 312–321. ACM. 62

Castagna, G., G. Ghelli, and G. Longo (1995). A calculus for overloaded functions with subtyping. *Inf. Comput. 117*(1), 115–135. 118

Chalin, P. and P. James (2007). Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP.* 119

Chambers, C. (1992). Object-oriented multi-methods in Cecil. In *ECOOP '92.* 6, 9, 57, 63, 117, 118

Chambers, C. and the Cecil Group (2004). The Cecil language: specification and rationale, Version 3.2. Available at http://www.cs.washington.edu/research/projects/cecil/. 55, 56, 57, 63, 73, 114, 115, 117, 118

Chiles, W. (2007, October). CLR inside out: IronPython and the dynamic language runtime. Available at http://msdn.microsoft.com/en-us/magazine/cc163344.aspx. Accessed 8/09. 125

Clifton, C., G. T. Leavens, C. Chambers, and T. Millstein (2000). MultiJava: modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00*, pp. 130–145. 6, 9, 33, 50, 63

Clifton, C., T. Millstein, G. T. Leavens, and C. Chambers (2006). MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst. 28*(3), 517–575. 5, 16, 17, 27, 106, 117

Cook, W., W. Hill, and P. Canning (1990). Inheritance is not subtyping. In *POPL*, pp. 125–135. 73

Coppo, M. and M. Dezani-Ciancaglini (1978). A new type assignment for λ-terms. *Archive for Mathematical Logic 19*(1), 139–156. 4, 52, 113

Coppo, M., M. Dezani-Ciancaglini, and P. Salle (1979). Functional Characterization of Some Semantic Equalities inside Lambda-Calculus. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pp. 133–146. Springer. 4, 52, 113

Davies, R. (2005, May). *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA. CMU-CS-05-110.

Davies, R. and F. Pfenning (2000). Intersection types and computational effects. In *ICFP '00*, pp. 198–208. 40

Day, M., R. Gruber, B. Liskov, and A. Myers (1995). Subtypes vs. where clauses: constraining parametric polymorphism. In *OOPSLA '95*, pp. 156–168. 114

DeLine, R. and M. Fähndrich (2004). Typestates for objects. In *ECOOP '04*, pp. 465–490. 127

Dreyer, D. (2005, May). *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA. CMU-CS-05-131. 55

Dreyer, D., K. Crary, and R. Harper (2003). A type system for higher-order modules. In *POPL '03*, pp. 236–249. 55

Dreyer, D. and A. Rossberg (2008). Mixin' up the ML module system. *SIGPLAN Not. 43*(9), 307–320. 55

Ducasse, S., O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black (2006). Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst. 28*(2), 331–388. 6, 54, 61, 62, 76, 90, 113

Duggan, D. and F. Bent (1996). Explaining type inference. *Sci. Comput. Program. 27*(1), 37–83. 126

Dunfield, J. (2007, August). *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University. CMU-CS-07-129.

Dunfield, J. and F. Pfenning (2004). Tridirectional typechecking. In *POPL '04*, pp. 281–292.

Ellis, M. and B. Stroustrup (1990). *The Annotated C++ Reference Manual.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. 6, 10, 61, 63, 64

Feinberg, N., S. E. Keene, R. O. Mathews, and P. Withington (1997). *The Dylan Programming Book.* Addison-Wesley Longman. 117

Findler, R. and M. Flatt (1999). Modular object-oriented programming with units and mixins. *ACM SIGPLAN Notices 34*(1), 94–104. 54, 90

Fisher, K. and J. Reppy (1999). The design of a class mechanism for moby. In *PLDI '99*, pp. 37–49. 7, 55, 93, 113

Fisher, K. and J. Reppy (2002). Inheritance-based subtyping. *Inf. Comput. 177*(1), 28–55. 113

Fisher, K. and J. Reppy (2004, January). A typed calculus of traits. In *Proceedings of the 11th Workshop on Foundations of Object-oriented Programming.* 54, 90, 113

Flatt, M., R. Findler, and M. Felleisen (2006). Scheme with classes, mixins, and traits. In *APLAS*, pp. 270–289. Springer. 6

Flatt, M., S. Krishnamurthi, and M. Felleisen (1998). Classes and mixins. In *POPL '98*. 6, 54, 61, 90

Forster, F. (2006). Cost and benefit of rigorous decoupling with context-specific interfaces. In *PPPJ '06*, pp. 23–30. 101, 119

Freeman, T. and F. Pfenning (1991). Refinement types for ML. In *PLDI '91*, pp. 268–277.

Frost, C. and T. Millstein (2006, January). Modularly typesafe interface dispatch in JPred. In *FOOL/WOOD'06*. 7, 63, 64, 65, 90

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. 5, 27, 50

Gil, J. and I. Maman (2005). Micro patterns in Java code. In *OOPSLA '05*, pp. 97–116. 119

Gil, J. and I. Maman (2008). Whiteoak: Introducing structural typing into Java. In *OOPSLA*. 55, 100, 110, 114, 124

Gregor, D., J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine (2006, October). Concepts: Linguistic support for generic programming in C++. In *Proceedings of OOPSLA '06*, pp. 291–310. ACM Press. 114, 126

Hall, C., K. Hammond, S. Peyton Jones, and P. Wadler (1996). Type classes in haskell. *ACM Trans. Program. Lang. Syst. 18*(2), 109–138. 56, 57

Harper, R. and M. Lillibridge (1994, January). A type-theoretic approach to higher-order modules with sharing. In *POPL*, pp. 123–137. 115

Harper, R. and C. Stone (2000). A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte (Eds.), *Proof, Language, and Interaction: Essays in Honor of Robin Milner.* MIT Press. 115

Hutchinson, N. C. (1987). *EMERALD: An object-based language for distributed programming.* PhD thesis, University of Washington, Seattle, WA, USA. 73

Igarashi, A., B. Pierce, and P. Wadler (1999, November). Featherwieght Java: a Minimal Core Calculus for Java and GJ. In *OOPSLA '99*. 38

Johnsen, E., O. Owe, and I. Yu (2006). Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci. 365*(1), 23–66. 73

Keene, S. and D. Gerson (1989). *Object-oriented programming in Common Lisp: a programmer's guide to CLOS*. Massachusetts. 6

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold (2001). An overview of AspectJ. In *ECOOP '01*, pp. 327–353. 115

Kim, T., K. Bierhoff, J. Aldrich, and S. Kang (2009). Typestate protocol specification in jml. In *SAVCBS '09*, pp. 11–18. 127

Laufer, K., G. Baumgartner, and V. Russo (2000). Safe structural conformance for Java. *The Computer Journal 43*(6), 469–481. 114

Leavens, G., A. Baker, and C. Ruby (2006). Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes 31*(3), 1–38. 127

Leavens, G. and T. Millstein (1998). Multiple dispatch as dispatch on tuples. *ACM SIGPLAN Notices 33*(10), 374–387. 117

Leino, K. (2007). Specifying and verifying software. In *ASE '07*, pp. 2. 127

Leroy, X., D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon (2004). The Objective Caml system, release 3.09. Available at http://caml.inria.fr/pub/docs/manual-ocaml/index.html. 7, 55, 93, 113

Liskov, B. (1983). *CLU Reference Manual*. Secaucus, NJ, USA: Springer-Verlag. 114

Liskov, B., D. Curtis, M. Day, S. Ghemawhat, R. Gruber, P. Johnson, and A. Myers (1994, February). Theta reference manual. 73, 114

Liskov, B., A. Snyder, R. Atkinson, and C. Schaffert (1977). Abstraction mechanisms in CLU. *Commun. ACM 20*(8), 564–576. 114

Liskov, B. and J. Wing (1993). Specifications and their use in defining subtypes. In *OOPSLA '93*, pp. 16–28. 114

Litvinov, V. (1998). Contraint-based polymorphism in Cecil: towards a practical and static type system. In *OOPSLA '98*, pp. 388–411. 114

Litvinov, V. (2003). *Constraint-Bounded Polymorphism: an Expressive and Practical Type System for Object-Oriented Languages*. PhD thesis, University of Washington. 57, 59, 114

Madsen, O. L. and B. Moller-Pedersen (1989). Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89*, pp. 397–406. 118

Magnusson, B. (1991, November). Code reuse considered harmful. *Journal of Object-Oriented Programming 4*(3). 8, 103

Malayeri, D. (2009a). *Coding Without Your Crystal Ball: Unanticipated Object-Oriented Reuse*. PhD thesis, Carnegie Mellon University, Amazon gift card code: 5QPG-26Q9CW-BXC3. 7, 61, 93

Malayeri, D. (2009b, January). CZ: Multiple inheritance without diamonds. In *FOOL '09*. 61

Malayeri, D. (2009c, August). Personal communication. 50

Malayeri, D. and J. Aldrich (2007, January). Combining structural subtyping and external dispatch. In *FOOL/WOOD '07*. 15

Malayeri, D. and J. Aldrich (2008a, July). Integrating nominal and structural subtyping. In *ECOOP 2008*. 8, 15, 47, 103

Malayeri, D. and J. Aldrich (2008b, May). Integrating nominal and structural subtyping. Technical Report CMU-CS-08-120, Carnegie Mellon University. 22, 47, 59

Malayeri, D. and J. Aldrich (2009a, October). CZ: Multiple inheritance without diamonds. In *OOPSLA '09*. 61

Malayeri, D. and J. Aldrich (2009b, March). Is structural subtyping useful? An empirical study. In *ESOP '09*. 93

Mandelbaum, Y., D. Walker, and R. Harper (2003). An effective theory of type refinements. In *ICFP '03*, pp. 213–225. 127

Meyer, B. (1992). *Eiffel: The Language*. Prentice Hall. 6, 10

Meyer, B. (1997). *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall. 61, 63, 64

Meyers, S. (1992). *Effective C++: 50 specific ways to improve your programs and designs*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA. 69

Microsoft (2009a). C# future. Available at http://code.msdn.microsoft.com/csharpfuture. Accessed 8/09. 125

Microsoft (2009b). Dynamic language runtime. Available at http://www.codeplex.com/dlr. Accessed 8/09. 125

Millstein, T. (2003). *Reconciling software extensibility with modular program reasoning*. PhD thesis, University of Washington, Seattle, WA, USA. 2, 5, 16, 17, 27, 64

Millstein, T., C. Bleckner, and C. Chambers (2002). Modular typechecking for hierarchically extensible datatypes and functions. In *ICFP '02*. 33, 53, 64, 117

Millstein, T., C. Bleckner, and C. Chambers (2004). Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst. 26*(5), 836–889. 17, 33, 53, 117

Millstein, T. and C. Chambers (2002). Modular statically typed multimethods. *Inf. Comput. 175*(1), 76–118. 2, 7, 17, 33, 63, 65, 117

Millstein, T., M. Reay, and C. Chambers (2003). Relaxed MultiJava: balancing extensibility and modular typechecking. In *OOPSLA '03*, pp. 224–240. 116

Milner, R., M. Tofte, R. Harper, and D. Macqueen (1997). *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press. 55, 114

Muschevici, R., A. Potanin, E. Tempero, and J. Noble (2008, October). Multiple dispatch in practice. In *OOPSLA 08*. 119

Musser, D. and A. Stepanov (1989). Generic programming. In P. Gianni (Ed.), *ISAAC '88*, Volume 38 of *Lecture Notes in Computer Science*, pp. 13–25. Springer. 96

Nelson, G. (Ed.) (1991). *Systems programming with Modula-3*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. 7, 114

Nystrom, N., S. Chong, and A. Myers (2004). Scalable extensibility via nested inheritance. In *OOPSLA '04*, pp. 99–115. 118

Nystrom, N., X. Qi, and A. C. Myers (2006). J&: nested intersection for scalable software composition. In *OOPSLA '06*, pp. 21–36. 118

Odersky, M. (2007). The Scala language specification. Available at http://www.scala-lang.org/docu/files/ScalaReference.pdf. 6, 10, 55, 56, 62, 64, 66, 69, 114, 118, 127

Odersky, M. and M. Zenger (2005). Scalable Component Abstractions. In *OOPSLA '05*. 10, 54, 55, 61, 62, 66, 79, 90, 114

Ostermann, K. (2008). Nominal and Structural Subtyping in Component-Based Programming. *Journal of Object Technology 7*(1). 3, 8, 115, 116, 125

Paepcke, A. (1993). *Object-Oriented Programming: The CLOS Perspective.* The MIT Press. 9, 117

Pierce, B. (2002). *Types and Programming Languages*. MIT Press. 8

Pierce, B. (2004). *Advanced Topics in Types and Programming Languages.* The MIT Press. 114

Pirkelbauer, P., Y. Solodkyy, and B. Stroustrup (2007). Open multi-methods for C++. In *GPCE '07*, pp. 123–134. 119

Pottinger, G. (1980). A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 561–577. 4, 52, 113

Raj, R., E. Tempero, H. Levy, A. Black, N. Hutchinson, and E. Jul (1991). Emerald: A general-purpose programming language. *Software: Practice and Experience 21*(1). 73

Reppy, J. and A. Turon (2007, July-August). Metaprogramming with traits. In *ECOOP '07*. 6, 114

Sakkinen, M. (1989). Disciplined inheritance. In *ECOOP*, pp. 39–56. 61

Schärli, N., S. Ducasse, O. Nierstrasz, and A. Black (2003). Traits: Composable Units of Behaviour. In *ECOOP '03*. Springer. 6, 66, 78, 79

Shalit, A. (1997). *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley. 6, 9, 117

Singh, G. (1994). Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess. 5*(1), 34–43. 61, 63

Smith, C. and S. Drossopoulou (2005). Chai: Traits for Java-like languages. In *ECOOP '05*. 6

Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA*, pp. 38–45. 61, 63, 79

Steele, Jr., G. L. (1990). *Common LISP: The Language* (Second ed.). Digital Press. 117

Steimann, F. (2007). The infer type refactoring and its use for interface-based programming. *Journal of Object Technology 6*(2). 101, 119, 126

Stoutamire, D. and S. Omohundro (1996). The Sather 1.1 Specification. Technical Report Technical Report TR-96-012, International Computer Science Institute. 116

Strom, R. and S. Yemini (1986). Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng. 12*(1), 157–171. 127

Stuckey, P., M. Sulzmann, and J. Wazny (2003). Interactive type debugging in haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pp. 72–83. 126

Sun Microsystems (2003). Java collections API design FAQ. Available at http://java.sun.com/j2se/1.4.2/docs/guide/collections/designfaq.html. 29, 30, 102

Szyperski, C., S. Omohundro, and S. Murer (1993). Engineering a programming language: The type and class system of Sather. In J. Gutknecht (Ed.), *Programming Languages and System Architectures*, Volume 782 of *Lecture Notes in Computer Science*. Springer. 69, 73, 116

Tempero, E. D., J. Noble, and H. Melton (2008). How do Java programs use inheritance? An empirical study of inheritance in Java software. In *ECOOP '08*, pp. 667–691. 98, 119

Thorup, K. K. (1997). Genericity in Java with virtual types. In *ECOOP*, pp. 444–471. 118

Tip, F. and T. B. Dinesh (2001). A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol. 10*(1), 5–55. 126

Vlissides, J. (1999). Visitor in frameworks. *C++ Report 11*(10), 40–46. 5

Wadler, P. and S. Blott (1989). How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pp. 60–76. 56

Wand, M. (1986). Finding the source of type errors. In *POPL '86*, pp. 38–43. 126

Warth, A., M. Stanojević, and T. Millstein (2006). Statically scoped object adaptation with expanders. In *OOPSLA*, pp. 37–56. 56

Washburn, G. (2008, December). Personal communication. 65, 66

Wehr, S., R. Lämmel, and P. Thiemann (2007, July). JavaGI: Generalized Interfaces for Java. In *ECOOP '07*. 3, 55, 56, 116, 119

Wehr, S. and P. Thiemann (2009). JavaGI in the battlefield: Practical experience with generalized interfaces. In *GPCE '09*. 116, 119